Escuela
Politécnica
Superior

Glyph and Position Classification of Music Symbols in Early Manuscripts

Degree in Computer Engineering

Final Degree Project

Author:
Alicia Núñez Alcover

Tutor/s:
Pedro J. Ponce de León
Jorge Calvo Zaragoza

Universitat d'Alacant
Universidad de Alicante

September 2019

Glyph and Position Classification of Music Symbols in Early Music Manuscripts

Author
Alicia Núñez Alcover (Student)

Directors
Pedro José Ponce de León (Tutor)
Department of Software and Computing Systems
Jorge Calvo Zaragoza (Co-tutor)
Department of Software and Computing Systems

Degree in Computer Engineering

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, September 3, 2019

Abstract

In this research, we study how to classify of handwritten music symbols in early music manuscripts written in white Mensural notation, a common notation system used since the fourteenth century and until the Renaissance. The field of *Optical Music Recognition* researches how to automate the reading of musical scores to transcribe its content to a structured digital format such as *MIDI*. When dealing with music manuscripts, the traditional workflow establishes two separate stages of detection and classification of musical symbols. In the classification stage, most of the research focuses on detecting musical symbols, without taking into account that a musical note is defined in two components: glyph and its position with respect to the staff. Our purpose will consist of the design and implementation of architectures in the field of *Deep Learning*, using *Convolutional Neural Networks* (CNNs) as well as its evaluation and comparison to determine which model provides the best performance in terms of efficiency and precision for its implementation in an interactive scenario.

Acknowledgments

I would like to thank my tutors Pedro J. Ponce de León and Jorge Calvo-Zaragoza for bringing me this idea of doing a research related with Deep Learning and the opportunity of writing a research paper. Also, I am grateful for always having their door opened whenever I ran into a doubt or problem about my research.

I also would like to thank my family and friends, who have always supported me and without them, I wouldn't be where I am today.

We can build a much brighter future
where humans are relieved of menial work
using AI capabilities.

Andrew Ng.

Contents

1 Introduction **1**
1.1 Motivation . . . 3
1.2 Objectives . . . 4
1.3 Project Structure . . . 5

2 State of the art **7**
2.1 Introduction . . . 7
2.2 Essential Musical Terminology . . . 7
2.3 Convolutional Neural Networks . . . 9
2.3.1 The Feature Extraction Stage . . . 10
2.3.2 The Classification Stage . . . 15
2.4 Regularization . . . 15
2.5 Evaluating Models . . . 16

3 Technologies **18**
3.1 Python . . . 18
3.1.1 Tensorflow . . . 18
3.1.2 Keras . . . 19
3.1.3 Pandas . . . 19
3.1.4 Matplotlib and Seaborn . . . 19
3.1.5 Numpy . . . 20
3.2 OpenCV . . . 20
3.3 Jupyter Notebook . . . 20
3.4 Google Colaboratory . . . 21
3.5 MuRET . . . 21

4 Methodology **22**
4.1 Introduction . . . 22
4.2 Label Data . . . 23
4.3 Data Preparation and Preprocessing . . . 25
4.3.1 Establish Input Scheme . . . 25
4.3.2 Data Preparation . . . 27
4.3.3 Data Preprocessing . . . 28
4.4 Architectural Designs . . . 29
4.4.1 Independent Glyph and Position Model . . . 30

4.4.2 Category Output Model 31
4.4.3 Category Output, Multiple Inputs Model 32
4.4.4 Multiple Outputs Model 32
4.4.5 Multiple Inputs and Outputs Model 33
4.5 Experimentation Design 33

5 Implementation 35
5.1 Introduction 35
5.2 Input Scheme 35
5.3 Convolutional Neural Networks 37
5.3.1 Functional API 37
5.3.2 Proposed Architectures 38
5.4 Experimentation 44
5.4.1 Evaluation Metric 45
5.4.2 Weights Dictionary 46
5.4.3 K-fold Cross-Validation 46

6 Experiments 48
6.1 Analyzing the Dataset 48
6.2 Cross-Validation Results 52
6.3 Discussion 55

7 Conclusion 57
7.1 Summary 57
7.2 Evaluation 57
7.3 Further Works 58

Bibliography 61

List of Figures

1.1 Artificial intelligence taxonomy. 1
1.2 Deep learning classification representation. 3

2.1 A sample of a music manuscript written in White Mensural notation. . . . 7
2.2 A staff with lines and spaces indicated. 8
2.3 Example of handwritten music symbols in white Mensural notation, showing its glyph and position. 8
2.4 Classification of two images by a convnet. Source [11]. 9
2.5 A typical convnet architecture. 10
2.6 A basic convolutional layer diagram. 10
2.7 An image of a cat as input data. Source [15]. 12
2.8 Feature maps of the first convolutional layer of each block. Source [15]. . 12
2.9 A convolution with stride 2. 13
2.10 A convolution with stride 2 and padding 1. 13
2.11 Left: Image before applying ReLU. Right: Image after applying ReLU. Source [17]. 14
2.12 An example of pooling with filter of size 2 and stride 2. 14
2.13 A five-fold cross-validation. 16

4.1 Methodology steps. 22
4.2 An unlabeled music manuscript written in Mensural Notation. 23
4.3 A sample of a music score labeled and encoded in agnostic grammar. . . . 23
4.4 A sample of strokes draw and its correspondent bounding boxes. 24
4.5 Control panel of MuRET. 24
4.6 A generated music score. 24
4.7 Data exploration and preprocessing workflow. 25
4.8 A sample of a Mensural notation staff. 26
4.9 Left: Glyph bounding boxes. Right: Enlarged bounding boxes. 26
4.10 A sequential convnet model. 29
4.11 A multi-input convnet model. 29
4.12 A multi-output convnet model. 30
4.13 Top: Independent glyph classification model. Bottom: Independent position classification model. 31
4.14 Category output model: enlarged inputs are provided as input and predicts a combined label. 31

4.15 Category output with multiple inputs: both glyph bounding box and enlarged images are provided as input and the model must predict a label from the Cartesian product of glyphs and positions. 32
4.16 Multiple outputs model: enlarged images are provided as input and the model must predict both the glyph and the position separately. 32
4.17 Multiple inputs and outputs model: both glyph bounding boxes and enlarged images are provided as input and the model must predict both the glyph and the position separately. 33
4.18 Experimentation design workflow. 33

6.1 Histogram of distribution of position classes. 49
6.2 Histogram of distribution of glyph classes, in logarithmic scale for better visualization. 50
6.3 Heatmap between glyph and position limited to a thousand. 51

Index of Tables

4.1 Quantity of images per manuscript. 27
4.2 Example data structure used for storing the images and targets. 27

5.1 Summary representation of the architecture example. 38
5.2 Summary representation of the sequential architectures. 40
5.3 Summary representation of the architecture multiple outputs model. . . . 41
5.4 Summary representation of the architecture category output, multiple inputs model. 43
5.5 Summary representation of the architecture multiple inputs and outputs model. 44

6.1 Old distribution of classes and samples in the Mensural notation symbols dataset. 48
6.2 New distribution of classes and samples in the Mensural notation symbols dataset. 49
6.3 Average of validation accuracy and loss (average ± std. deviation) results achieved by a 5-fold cross-validation scheme. 52
6.4 Validation accuracy and loss (average ± std. deviation) results of glyph and position achieved by a 5-fold cross-validation scheme in multi-outputs models. 53
6.5 Accuracy results per each fold in CV approach with respect to the neural architecture considered for music symbol classification. 54
6.6 Accuracy (average ± std. deviation) and complexity with respect to the neural architecture considered for music symbol classification. The complexity of each model is measured as millions of trainable parameters. . . 55

1 Introduction

Over the past few years, artificial intelligence (AI) has been a thriving subject of huge success in a variety of application domains and research topics. AI research is going forward at an accelerated rate. Terms such as machine learning, deep learning, and AI come up in numerous scientific articles and magazines. This brings up the question of whether AI technologies will shape our future in the upcoming years. We have already seen a peek with the emergence of some AI applications such as self-driving cars.

We have always dreamed a future of building intelligent machines – our own robotic personal assistants or self-driving cars. Perhaps a future doomed for some and desired for others. The fantasy of creating artificial life dates back to ancient Greece with *Hephaestus* and *Pygmalion* who incorporated this idea with the tales of golden robot *Talos* or artificial *Pandora*, being AI a new essential element to myth and science fiction until nowadays.

First and foremost, we need to define what we are talking about when we mention AI. The terms AI, machine learning, and deep learning have been used interchangeably despite the fact that they are not the same.

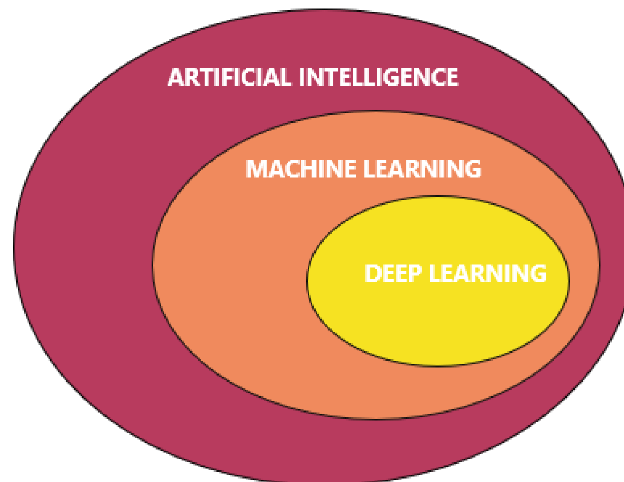

Figure 1.1: Artificial intelligence taxonomy.

AI could be narrowed to the definition of *the capability of performing human-like tasks.* Therefore, we can determine that AI is a field which includes machine learning and deep learning among other subfields as shown in Fig. 1.1. Machine learning appeared when the question: *could a computer learn on its own?* arose, creating a new paradigm where machines could learn by themselves using human input data and produce the desired output through a mathematical formula by feeding the machine learning system relevant examples, then allowing the system come up with rules for automating the task. For instance, if you wished to predict bitcoin prices it could be easily done by using a linear regression algorithm or if you wanted to program a machine that learns how to filter spam from your e-mails it could be done with algorithms such as Naive Bayes.

Going further, in machine learning there are different types of learning:

- **Supervised learning.** It is the most dominant type of machine learning. Given a labeled dataset, it produces a model that allows inferring the label of an unknown example. The goal of a supervised learning algorithm is to produce a model capable of approximating the correct mapping function.

- **Unsupervised learning.** It assumes an unlabeled dataset where there is no expected output. The goal of an unsupervised learning algorithm is to learn unknown patterns about the data.

- **Reinforcement learning.** An agent receives signals and decides actions in order to maximize a reward by trial-and-error. DeepMind applied this algorithm successfully in Atari games such as Breakout and Star Gunner.

Following the taxonomy of AI, deep learning is a specific subset of machine learning, a way of learning from data based on successive layers of representations learned by the so-called *Artificial Neural Networks* (ANN). The deep in deep learning stands by the idea of multiple levels of layers which contributes to the depth of the model or network. These neural networks were inspired by how information is processed in our brains through the connections of our neurons. As shown in Fig.1.2, this is a classification task's example using a deep learning approach.

How these layers operate in the prediction of a given input is stored in the weights (parameters) of a layer. Afterward, the final output will be determined by the final weights. To control the final output, a loss function appears: it will take the predictions and compare them to the real output to calculate how well the network is performing.

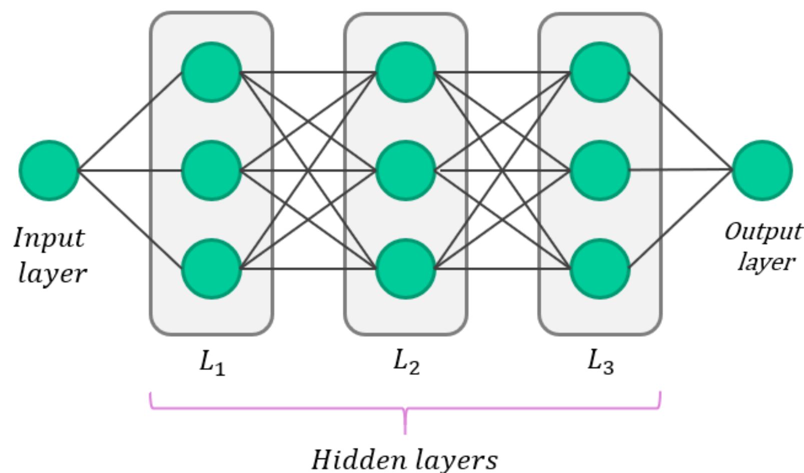

Figure 1.2: Deep learning classification representation.

1.1 Motivation

Music is an essential art form for our lives, it has been a trait in different types of culture throughout the course of human history. For centuries, composers would share their art by penning it on paper with ink, known as musical manuscripts. A considerable amount of these musical manuscripts are preserved in historical archives, which are usually transcribed in a digital format for ease of use but in order to exploit their usefulness, it is necessary to transcribe these sources to a structured format such as MusicXML [1], MEI [2], or MIDI. Until now, it has only been done manually by professionals in slow and expensive processes. Thanks to Optical Music Recognition (OMR) techniques, which its goal is to decode sheet music and create new scores in a machine-readable format, it is easier to automate the process of reading music notation from scanned music scores [3].

The transcription of historical musical documents is treated differently with respect to traditional OMR methods due to particular characteristics of these manuscripts, such as the use of certain notation systems. Although there exist several works focused on early music documents transcription [4, 5], the specificity of each type of manuscript, or its overall writing style makes it difficult to generalize these developments.

In this context, a music transcription system is one that performs the task of obtaining a digital structured representation of the musical content in a scanned music manuscript. The workflow to accomplish such a task could be summarized as follows: First, a document layout analysis step isolates document parts containing music, mostly music staffs. Then, an OMR system detects music symbols contained in these parts, typ-

ically producing a sequence or graph of music symbols and their positions with respect to the staff. From this representation, a semantic music analysis step assigns musical meaning to each symbol, as this often depends on the specific location of the symbol in the sequence. Finally, this intermediate music representation is translated by a coding stage into a structured representation in the desired output format.

Unlike other domains, in the particular case of music notation, the symbols to be classified have two components: glyph and position in the staff. Traditionally, OMR systems use supervised learning to predict the glyph [6, 7, 8], whereas the position is determined by heuristic strategies [9]. Since these OMR systems usually perform a pre-process that normalizes the input images such as binarization, these heuristic strategies tend to be quite reliable. However, other approaches might use different pre-processing steps, and so traditional heuristics might not work correctly. This is why we propose to deal with the identification of the glyph position by means of supervised learning as well.

To this end, our purpose in this research is to study the best approach to perform classification of a pre-segmented symbol image into its pair of glyph and position. Specifically, we propose different deep learning architectures that make use of Convolutional Neural Networks in order to fully classify a given music symbol. We aim to analyze which architecture gives us the best performance in terms of accuracy and efficiency.

1.2 Objectives

This project sheds new light on the classification of handwritten music scores in mensural notation, used from the fourteenth century until the Renaissance. This could be achieved thanks to the research done in the design and analysis of different developed classification schemes in order to be implemented in an interactive scenario.

To complete our research it is intended to develop the objectives as follows:

- **Labeling a corpus of handwritten music scores.** As a first objective, we will contribute by labeling a set of images of a given corpus in mensural notation using pen-based technologies and transcription tools.

- **Exploration, preparation and pre-processing of the corpus.** An important step in any deep learning related workflow is to explore our raw data, analyze it and transforming it into clean data to be used later to feed our models.

- **Design of different classification schemes to classify music symbols.** We are going to design different approaches that perform the different classification

stages either simultaneously or independently.

- **Implementation of the classification schemes using Convolutional Neural Networks.** After designing our architectures, we will implement them by creating architectures using Convolutional Neural Networks as the classification method.

- **Experimentation of the developed architectures.** Experiments will be carried out using our new clean data and developed architectures to assess the accuracy and effectiveness of our architectures.

- **Evaluation and comparison between the architectures.** Our final step will be to evaluate the results achieved by the experiments and compare between them to determine which one has the best efficiency and precision to be used in a real interactive scenario.

1.3 Project Structure

In order to facilitate its reading, the contents of this project are divided into chapters and sections. The project's structure used is organized as follows:

- ***Chapter 1: Introduction*** ⇒ Introductory chapter that covers the fundamentals of this project and describes the objectives intended to be achieved.

- ***Chapter 2: State of the art*** ⇒ Dedicated to providing a general theoretical basis of the area in the field that the problem is covered to be solved and its intention in this project.

- ***Chapter 3: Methodology*** ⇒ Explains the details of the workflow followed as well as the techniques needed to be carried out in order to develop powerful architectures for our research.

- ***Chapter 4: Technologies*** ⇒ Addresses the technologies used throughout the project such as the programming language and environment.

- ***Chapter 5: Implementation*** ⇒ Dedicated to explaining the details of the algorithms and structures implemented in the project.

- ***Chapter 6: Experiments*** ⇒ Shows the configuration and results of the experiments as well as the results obtained and its further analysis.

- ***Chapter 7: Conclusion*** ⇒ Contains a summary of the followed process during

the project, the thoughts obtained regarding the study of the results, as well as the introduction on some ideas for future research.

2 State of the art

2.1 Introduction

In order to fully understand this research, it is necessary to explain beforehand essential related terminology used throughout the course of this project, which will be explained in this section.

2.2 Essential Musical Terminology

A dataset of music manuscripts is going to be handled in this research. A music manuscript can be defined as a handwritten source of music which can contain musical notation as well as texts such as lyrics. Usually, the medium of music manuscripts is paper or papyrus like used in earlier centuries. The appearance of these music manuscripts dates back to the 9th century. In our case, we are dealing with music manuscripts written in *White Mensural Notation*, a common European system of musical notation used in the XVI and XVII centuries. An example is shown in Fig. 2.1.

Figure 2.1: A sample of a music manuscript written in White Mensural notation.

The main and first component of a music manuscript is, the **staff**, or also called stave, which consists of five horizontal lines in parallel, on which musical notes are placed as shown in Fig. 2.2. These lines and spaces represent different pitches or, how high or low a musical symbol is.

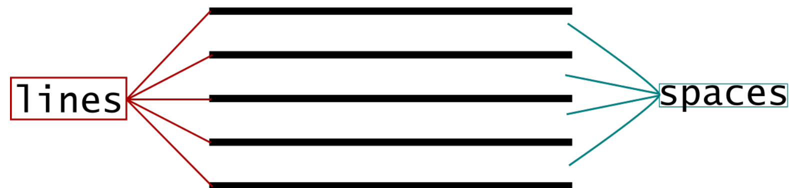

Figure 2.2: A staff with lines and spaces indicated.

We are going to consider that all musical symbols are defined by two components: **glyph** or graphic symbol, which can be described as the isolated element placed in a line or space of the staff, and **position** with respect to the staff lines. This is obvious in the case of notes, as these components indicate the duration and the pitch, respectively. We can generalize this to any type, as all symbols are located in a specific position with respect to the lines of the staff. Let $\mathcal{G}$ be the label space for the different glyphs and $\mathcal{P}$ the label space for the different positions. A music symbol is therefore fully defined by a pair $(g, p), g \in \mathcal{G}, p \in \mathcal{P}$. A graphical example is given in Fig. 2.3. Note that position labels refer to the vertical placement of a glyph: Ln and Sn denote symbol positions over or between staff lines, respectively.

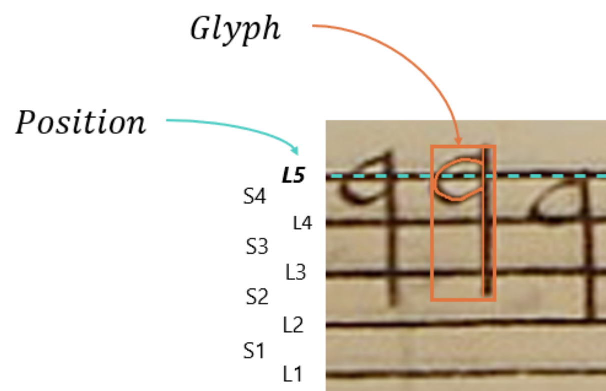

Figure 2.3: Example of handwritten music symbols in white Mensural notation, showing its glyph and position.

2.3 Convolutional Neural Networks

This section introduces Convolutional Neural Networks (CNN) or *convnets*, the most common technique used for computer vision and image recognition. Convnets have been proven successful in classification and recognition of images [10] such as objects, food or traffic signs. Fig. 2.4 illustrates an example of a convnet recognizing different scenes.

Figure 2.4: Classification of two images by a convnet. Source [11].

Convnets can be defined as *simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers* [12]. Yann LeCun, considered the father of convnets [13], created the first one, called LeNet, in the year 1994 which was used for character recognition tasks like digits. Nowadays there are a handful of architectures such as GoogleNet or AlexNet. How do these architectures work?

An image can be defined as a matrix of pixel values. Pixels of an image tells us useful information of what we are seeing: in the left image of Fig. 2.4, we can see grass, clothes and a frisbee. Most of the pixels that are side by side represents the same information – we can still see grass, although is darker in some pixels. Therefore, we can train neural networks to differentiate the patterns in a given image. This is when convnets come into play.

In order to create a convnet architecture we need three different type of layers: convolutional layer, pooling layer and fully-connected layer. A typical convnet architecture is shown in Fig. 2.5. Note that $Conv_n$ refers to convolutional layer, $Pool_n$ to pooling layer and FC_n to fully-connected layer.

As illustrated in the figure we can define two stages in a convnet architecture: **feature extraction** and **classification**, which will be explained in the following subsections.

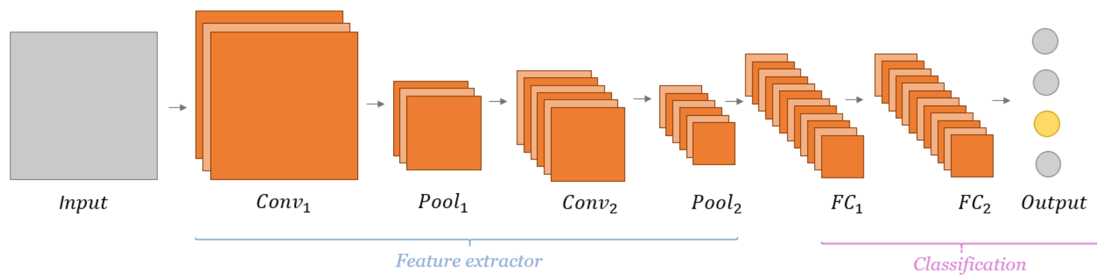

Figure 2.5: A typical convnet architecture.

2.3.1 The Feature Extraction Stage

In this stage, our convnet is going to learn features from the input data in order to differentiate patterns. The feature extractor is composed, usually, of convolutional layers and pooling layers. A typical convolutional layer encompasses three stages as shown in Fig. 2.6.

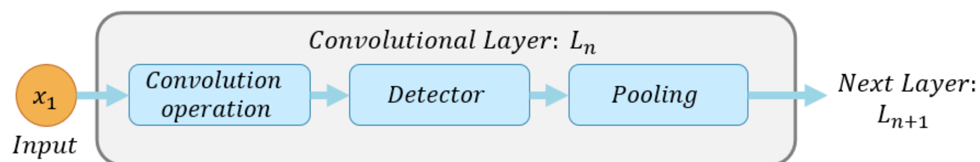

Figure 2.6: A basic convolutional layer diagram.

An input x_i passes through a convolutional layer L_n, where in the first step, the layer carries out convolution operations to produce linear activations. In the second step, each linear activation is run through the detector layer, which is a non-linear function and, finally, we use a pooling layer to downsample the produced output. Afterward, the produced output passes through the next layer L_{n+1}.

A convolutional layer learns local patterns, since the most useful information of any image is local. Therefore, it would be logical to think that we can create *squares* (or more widely called *windows*) that slides over an image and learn its patterns. These patterns are *translation invariant*: A pattern recognized in an upper-left corner can be recognized anywhere in a new image. Moreover, these patterns can be *hierarchically composed*: in the first layers of a network can identify lines and edges but as we go deeper in our architecture, it can identify more complex features [14].

A **convolution**, mathematically speaking, is an operation between two functions f and g that produces a third one by integration, expressing how the shape of one is modified. Equation 2.1 illustrates the convolution operation formula:

$$(f * g)(t) = \frac{\mathrm{d}}{\mathrm{d}t} \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau \tag{2.1}$$

A convolutional layer is basically a set of multiple convolution filters. Convolutions operate over *feature maps* with three spatial axes: *width*, *height* and *depth*. For instance, an image might have the size $5 \times 5 \times 3$ in the first convolutional layer, this means that has 5 pixels of height and width and 3 of depth – since it is an RGB image, it has three color channels; a black-and-white image would have 1 as depth. Then, the convolution operation slides – or convolves – each filter across the feature maps inputs (or patches) and checking if the feature it is meant to detect is present by computing dot products between the entries of the filter and the input in every position, producing an output feature map, which still has width and height, and an activation map that will activate the filters, which encode specific aspects of the input data.

As an example, we are going to detect a pattern. Consider an image I in grayscale (0 values represent white and 1 values represent black). Then, we extract a patch B of 3×3 size of I and a matrix F of 3×3 as the kernel (or filter), the convolution can be computed as shown below:

$$B = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad F = \begin{bmatrix} 5 & 2 & 3 \\ 0 & 4 & 3 \\ 0 & 3 & 0 \end{bmatrix} \tag{2.2}$$

The convolution $(B * F)$ would be as follows:

$$(B * F) = \begin{bmatrix} 1 \cdot 5 & 1 \cdot 2 & 1 \cdot 3 \\ 0 \cdot 0 & 1 \cdot 4 & 0 \cdot 3 \\ 0 \cdot 0 & 1 \cdot 3 & 0 \cdot 0 \end{bmatrix} = 17 \tag{2.3}$$

Basically, a filter is a feature detector and when the higher the number is, the more likely the pattern is to be present in the patch.

Feature maps learned by the layers can be visualized. For instance, we are going to visualize them using a trained network, given an image of a cat as shown in Fig. 2.7.

Figure 2.7: An image of a cat as input data. Source [15].

As a result, the feature maps that correspond to the first convolution of each block of a convnet with five blocks are shown in Fig. 2.8. Note that the figure only displays 8 feature maps for each layer for the sake of simplicity.

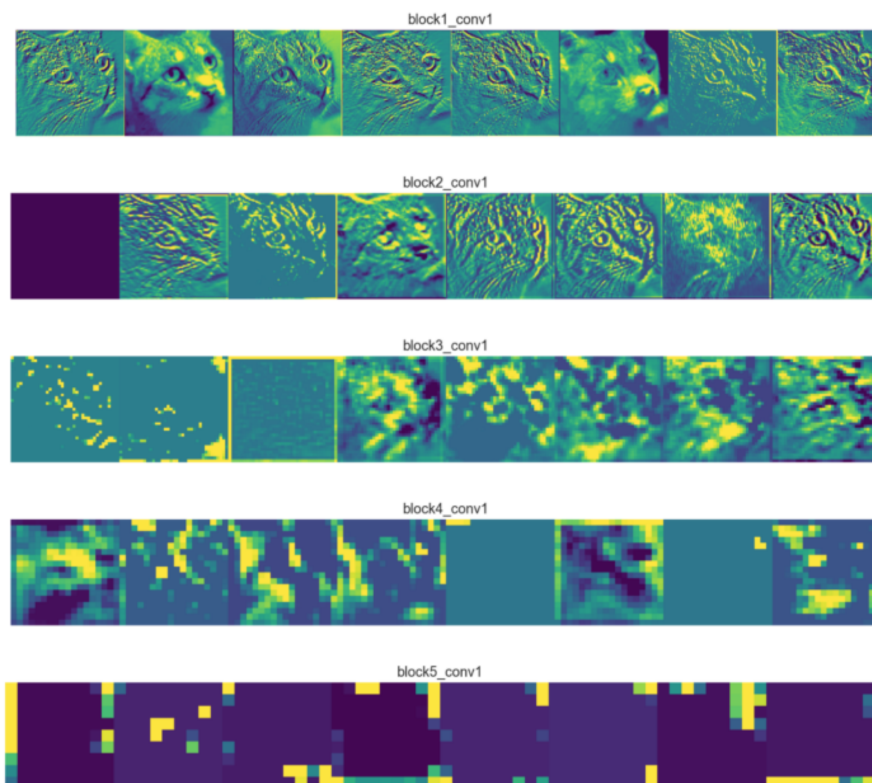

Figure 2.8: Feature maps of the first convolutional layer of each block. Source [15].

Each layer learns a set of filters. We can see that the first layers learn simple things such as edges. As we go deeper in the layers, the filters get increasingly complex and less visually understandable since they are encoding elaborated concepts such as ear or eye.

Two important properties of convolution are **stride** and **padding**. Stride can be defined as the step size of the moving window. If a stride size is 1, it means the filter slides pixel by pixel. As shown in Fig. 2.9 you can see that the output matrix produced is smaller, meaning that width and height are being downsampled by a factor of 2.

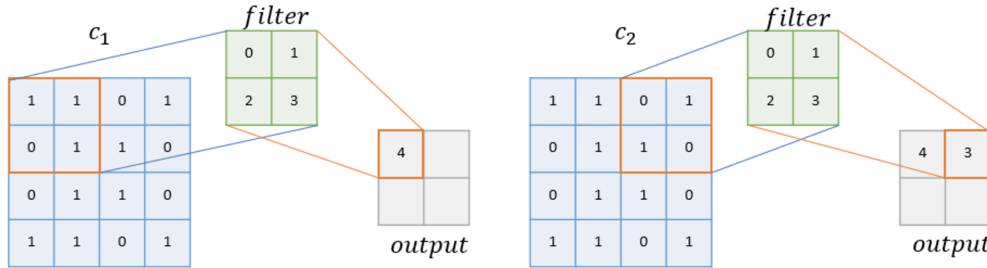

Figure 2.9: A convolution with stride 2.

Padding consists of adding rows and columns surrounding the image. These new tiles added normally contain zeroes. As shown in Fig. 2.10, you can see that the produced output matrix is bigger.

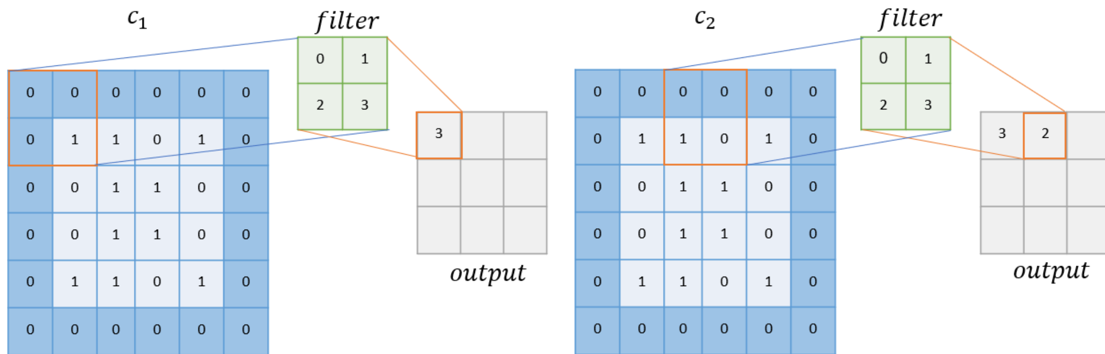

Figure 2.10: A convolution with stride 2 and padding 1.

Note that for the sake of simplicity, only two convolution operations are shown in Fig. 2.9 and Fig. 2.10.

The next **step** is the process of applying an activation function to the linear activations produced in the convolution operation. *Rectified Linear Unit* (ReLU) is the most common for convnets.

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{2.4}$$

ReLU will apply an elementwise operation, replacing all the negative pixel values in the feature map to zero as shown in Fig. 2.11. The main purpose is to introduce non-linearity in the convents. An advantage is the faster convergence, six times faster than *Tanh* and *Sigmoid* activations. As a disadvantage is that they are initially in the off-state as zero gradients, therefore weights will not be updated in the backpropagation. This is fixable, for example, by using the activation *Leaky ReLU* which forces small negative gradient through the network [16].

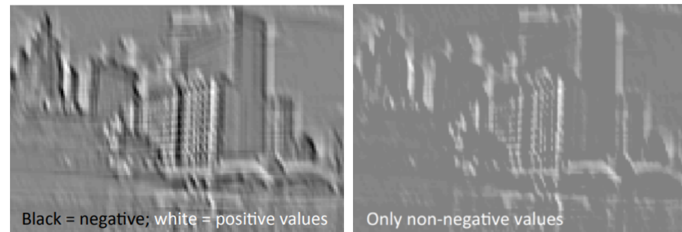

Figure 2.11: Left: Image before applying ReLU. Right: Image after applying ReLU. Source [17].

The final **step** is pooling, which usually follows a convolutional layer and gets the output of a convolution as its input. It is conceptually similar to convolution, as a filter applied using a moving window approach. Basically, applies a fixed operator such as max (as in max-pooling) or average, and downsamples feature maps by extracting squares from the input and outputting the max value.

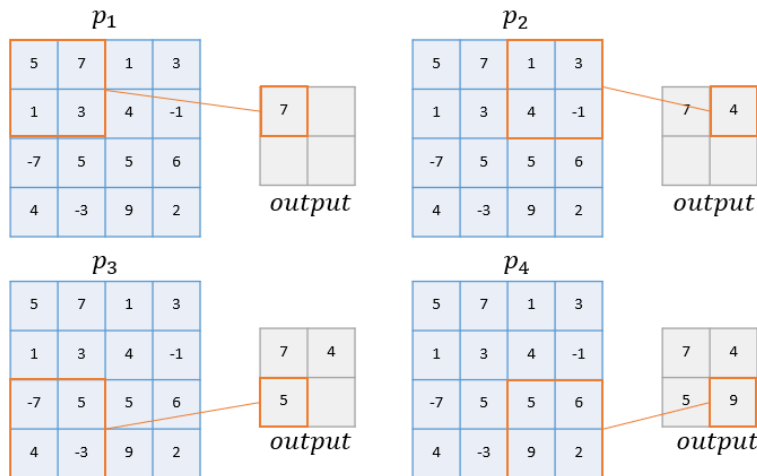

Figure 2.12: An example of pooling with filter of size 2 and stride 2.

Pooling contributes to the increasing of accuracy in a model, and speeds the training

by reducing parameters as you can see in the example given in Fig. 2.12 and p_n denotes the number of pooling operation. It is clearly shown that the number of parameters was reduced by 25%.

2.3.2 The Classification Stage

The classifier is composed, usually, by fully-connected layers where the input is flattened into a feature vector and passed through a network of neurons to predict the output probabilities.

Then, the next layer, which is the output layer, is responsible for producing the probability of the possible classes, given an input image. This layer passes through an activation, such as **Softmax**, that maps into a vector all the numbers into probabilities that sum to one.

$$\sigma(z_j) = \frac{e(z_j)}{\sum_{k=1}^{K} e(z_k))} \tag{2.5}$$

Once our model is completed, we need to select our **loss function** and **optimizer**. The former will be used to measure the accuracy of our network, such as *categorical cross-entropy* and the latter will determine how we update the weights of our network according to the loss function, like *stochastic gradient descent* (SGD). There are also variants of SGD like **Adam** or **RMSprop** [18].

2.4 Regularization

Before diving in regularization, there are two terms that we need to acknowledge before developing our models. The ability of a network to be able to predict correctly unseen data is called **generalization**. Therefore, we can state that our purpose –our only purpose– is to achieve a model capable of generalizing well. But, our models can suffer from:

- **Underfitting**. Or also called, **high bias**, is the inability of our model to predict correctly in training data as well as unseen data. This could be because our models are too simple or the features learned are not useful enough.

- **Overfitting**. Or **high variance** is when our models are able to generalizing *too well* in the training data but unable to predict correctly in our validation sets. In this case, this could be because our models are too complex or the training data is not enough.

Regularization is a useful technique to prevent overfitting by forcing the model to be less complex. One of several benefits from pooling layers is that they force the network to focus on just a few neurons, regularizing our network and thus be less likely to overfit. Moreover, we can find more regularization techniques: **Dropout** and **Batch-Normalization** [19].

Dropout is the most effective and used technique for regularization [20]. Basically consists in dropping out – or set to zero – a number of neurons with a probability $1 - p$ during training. Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsible for the inputs. This conceptualization suggests that perhaps dropout breaks-up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust [21].

There are also some explicit non-mathematical regularization methods such as **Data Augmentation**.

Data Augmentation is a technique that creates synthetic data based on the original data by applying transformations: rotating, flipping, cropping, zooming, and so on. The best way to prevent overfitting is to have more training data.

2.5 Evaluating Models

Evaluating a learning algorithm can be tricky. A simple way to evaluate a model is splitting the data into training and test set. Then, the training set would be used to train the architecture and testing set to test it. Nevertheless, this method can result in a biased estimation of the model skill. Cross-validation is a solution. The goal of cross-validation is to test how well the model will generalize on unseen data.

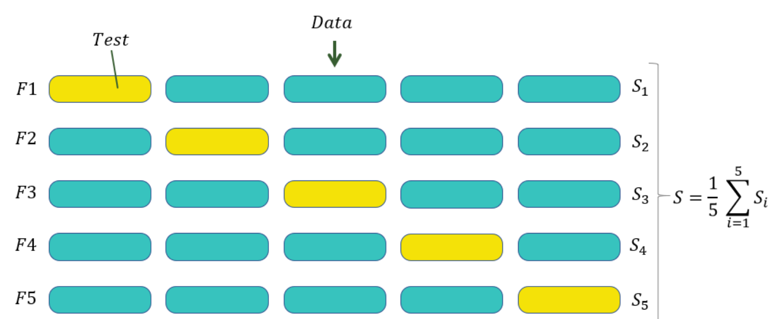

Figure 2.13: A five-fold cross-validation.

You split your training data into k folds of equal size. For example, with five folds you randomly split your training data into five folds: $\{F_1, F_2, F_3, F_4, F_5\}$. Each fold contains 20% of your training data. The k value has to be chosen wisely since there is a bias-variance trade-off associated with the choice of k [22].

To train the first model, f_1, you use all data from folds F_2, F_3, F_4, F_5 as training set and F_1 as the validation set. We would do the same for the rest of the folds. Then you average the five S_n values of the metric to get the final value S. Schematically, a k-fold cross-validation is shown in Fig. 2.13.

There are variations of cross-validation such as **Holdout method** or **Stratified**. For instance, in stratified k-fold cross-validation, the folds are selected so the number of each label is approximately equal in all the folds.

3 Technologies

In this section, we cover the technologies used in order to implement our approaches throughout our project such as programming language used, or which technologies have been used to analyze the data used.

3.1 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics and used to write the code of this project. Python is simple, easy to learn syntax emphasizes readability which reduces the cost of program maintenance. While complex algorithms and versatile workflows stand behind machine learning and AI, Python's simplicity allows developers to write reliable systems which makes it easier to build models for machine learning. Moreover, Python supports modules and packages, which encourages program modularity and code reuse. With its rich technology stack, has an extensive set of libraries for artificial intelligence and machine learning needed for our research, such as Keras, Tensorflow and Scikit-Learn, which will be explained in the following subsections.

3.1.1 Tensorflow

Tensorflow is an open-source artificial intelligence library, using data flow graphs to build models. It allows developers to create large-scale neural networks with many layers. It was originally developed by the Google Brain Team within Google's Machine Intelligence research organization for machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well. TensorFlow is cross-platform. It runs on nearly everything: GPUs and CPUs—including mobile and embedded platforms—and even tensor processing units (TPUs), which are specialized hardware to do tensor math on.

3.1.2 Keras

Keras is a high-level neural networks API, written in Python and supports multiple back-end neural network computation engines such as Tensorflow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research. Keras allows easy and fast prototyping through its user-friendliness, modularity and extensibility. Provides two ways to build models: sequential, which allows you to create models layer-by-layer and functional, which supports more flexibility as you can easily define models with shared layers or have multiple inputs or outputs.

Beyond its ease of learning and ease of model building, Keras supports both convolutional networks, needed for this project, and neural networks, as well as its combination. Plus, Keras has strong support for multiple GPUs and distributed training being able to run seamlessly on CPU and GPU.

3.1.3 Pandas

Pandas is an open-source software library created for the Python programming language ready for data manipulation and data analysis. This library provides high-performance, easy-to-use data structures, operations for manipulating numerical tables and time series.

Pandas uses DataFrame as its data structure, a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns), which will be useful to create our data structures needed for our project.

3.1.4 Matplotlib and Seaborn

Matplotlib and Seaborn are powerful visualization libraries which can be used in Python scripts. On the one hand, Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. On the other hand, Seaborn is another data visualization library based on Matplotlib, which provides a high-level interface for drawing attractive and informative statistical graphics.

3.1.5 Numpy

NumPy is the core library for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.

3.2 OpenCV

OpenCV is an open-source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code as well as its support for the deep learning frameworks TensorFlow, Torch/PyTorch and Caffe.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, among others.

OpenCV will be used to read the images and create the input scheme needed for the project, which will be explained in the Methodology section.

3.3 Jupyter Notebook

The Jupyter Notebook is an open-source web environment that allows you to create and share documents called notebook documents which can contain live code such as python and rich text elements like equations, visualizations and text. Therefore, it is a powerful tool for performing tasks in data cleaning, visualizing data statistical modeling and creating machine learning models.

We will use Jupyter Notebook to create our notebook containing the code of this project, alongside Google Colaboratory, which will be explained in the following subsection.

3.4 Google Colaboratory

Colaboratory is a Google research project created to help students and researchers in the machine learning field. It is a free environment based on Python Jupyter notebooks that requires no setup to use and runs entirely in the cloud provided by Google. Each session provides a VM with 25 GB of ram where you can use free GPUs such as Tesla K80, TPUs and CPUs, opening up the opportunity to let anyone practice deep learning using frameworks like TensorFlow, PyTorch, Keras and OpenCV.

You just need a Google Drive account and you can create notebooks, upload notebooks, store notebooks in your Google Drive and share them.

3.5 MuRET

MuRET [23] is a tool used for music recognition, encoding, and transcription. Covers all transcription phases, from the manuscript source to the encoded digital content. MuRET is designed as a technology-focused research tool, allowing different processing approaches to be used, and producing both the expected transcribed contents in standard encodings and data for the study of the transcription process itself.

We are going to use this tool for labelling a corpus of handwritten music scores using pen-based technologies.

4 Methodology

4.1 Introduction

In this chapter, we are going to address the methodology used throughout our research. Our workflow will be composed of four stages as shown in Fig. 4.1.

Figure 4.1: Methodology steps.

- **Label data.** Given a set of unlabeled music manuscripts of a dataset, needed for our data, we are going to use pen-based technologies and transcription tools to label them.

- **Data preparation and preprocessing.** A comprehensive data exploration is going to be made, in order to understand our variables as well as analyzing and preprocessing them. Moreover, we will establish an input scheme to feed our architectures.

- **Architectural designs.** In this step, we are going to design our different classification schemes using convnets based on the background that we established.

- **Experimentation design.** This step outlines how the experiments were designed in order to obtain the needed results to evaluate our models and then, decide which approach we should aim to.

4.2 Label Data

In this stage, we are going to be hands-on the dataset and we will label musical manuscripts by using the web application **MuRET**. As we explained in previous sections, these music manuscripts are written in White Mensural notation, a common notation used in the centuries XVI and XVII. An unlabeled music manuscript can be seen in Fig. 4.2.

Figure 4.2: An unlabeled music manuscript written in Mensural Notation.

Our purpose is to tag these musical symbols into its two components: *glyph* and *position*. Moreover, these musical symbols are encoded into what we call **agnostic grammar**. Consequently, a musical symbol is defined as *glyph:position*. For instance, if we have a whole note in line 6, it would be encoded as *note.whole:L6*. Fig. 4.3 illustrates a sample of a music score written in agnostic grammar.

Figure 4.3: A sample of a music score labeled and encoded in agnostic grammar.

These images are stored in the web application. We can label them using two different mediums: manually or through the use of classifiers based on Deep Learning's algorithm, which is our final purpose.

Therefore, we are going to label these images manually, through the help of pen-based technologies – a graphic tablet– to draw the strokes of the symbols. The aim is to obtain the correspondent bounding box of a given musical symbol and its coordinates by drawing its stroke. Once you draw it, the next step is to label its glyph and position.

Figure 4.4: A sample of strokes draw and its correspondent bounding boxes.

The next step would be to select the symbols manually and classify them. As we can see in Fig. 4.4, we have two options: *Move pitch*, that is, the position of the specific symbol and, on the right, choose the glyph to which it corresponds according to the different sections (Clef meters, Notes, Beamed notes…).

Figure 4.5: Control panel of MuRET.

As we classify them, the symbols will be generated in a new music score, as we can see in the Fig. 4.6.

Figure 4.6: A generated music score.

For instance, once the stroke of the second musical symbol in Fig. 4.4 was made, this symbol would be classified with the glyph: *note.half_down* category that we can find in the Notes section and the position would be *S4*.

As a final result, we would obtain a **JSON** file of the information and coordinates of each of the symbols that we have previously classified per image, which will be needed for later in order to train our models.

4.3 Data Preparation and Preprocessing

Figure 4.7: Data exploration and preprocessing workflow.

Fig.4.7 gives a brief overview of the workflow followed in this stage.

- **Establish input scheme.** First of all, we need to determine what inputs we will use to feed our models in order to achieve the most effective approach to classify musical symbols.
- **Data exploration and preparation.** In this step, we need to dive into our dataset and get a better understanding of it by analysing and visualizing it.
- **Data preprocessing.** Our final step consists in processing our raw data and converting it into clean data in order to be feasible to feed our models.

4.3.1 Establish Input Scheme

Our work assumes a **segmentation-based** approach, in which the locations of symbols that appear in the input music score have already been detected in a previous stage, as we did in the last section. This can be achieved under an interactive environment where the user manually locates the symbols [24], but can also be automated with object detection techniques [25].

Considering the above, the complete classification of a music-notation symbol consists in predicting both its glyph and its position. This opens up several possibilities as regards this dual process, given that the two components are not completely independent. As a base classification algorithm, we resort to convnets as aforementioned in order to learn a suitable data representation for the task at hand [26]. The classical use of a convnet is to consider a single image as input, that must be associated with a single class label. However, since we want to know the glyph of a symbol as well as its position within the staff simultaneously, we shall consider different architectures with shared layers, and multi-output and multi-input models, in order to determine which is the best way to obtain the corresponding full symbol classification.

Therefore, we need an approach of how to represent the input for classification as follows:

Two region-based image inputs are used in this approach. Fig 4.8 shows a part of a music staff from a Mensural notation score. These images are pre-segmented, either manually or automatically, by defining a bounding box around each music symbol in the staff, as shown in Fig. 4.9 (left). Thus, each bounding box defines an image instance containing a music symbol. These appropriately annotated images can be used to train and evaluate a music glyph classification model. However, in general these instances do not span vertically as to contain all staff lines. Therefore, they do not convey information about the music symbol position. For example, the symbol labeled as 'A' in Fig. 3 (left) is indistinguishable from symbol 'B' in the same image in spite of appearing at different staff positions. This type of images will be referred as *glyph inputs*.

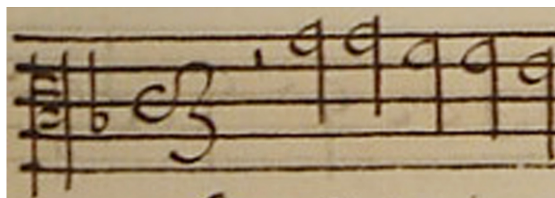

Figure 4.8: A sample of a Mensural notation staff.

In order to produce models able to correctly classify music symbol positions, a second image set is constructed by enlarging the bounding box frame vertically to a fixed height large enough to contain all staff lines, as shown in Fig. 4.9 (right). This type of images will be referred as *enlarged inputs* in the following sections. It has been shown that these enlarged images contain enough information for estimating the vertical position of a music symbol within the staff [27].

To achieve this task at hand, we apply *OpenCV* techniques to read and handle the images in dynamic programming mode. Furthermore, we store the image instances as *Numpy* arrays in a *Pandas DataFrame*.

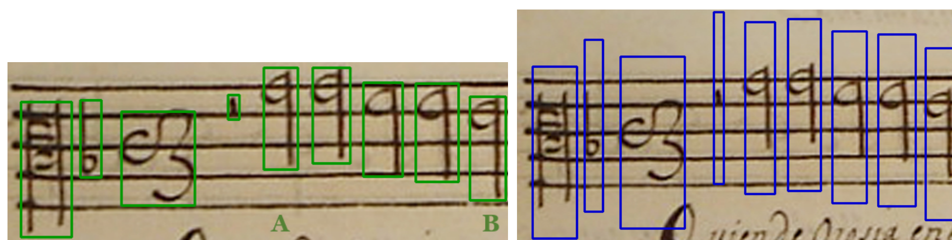

Figure 4.9: Left: Glyph bounding boxes. Right: Enlarged bounding boxes.

4.3.2 Data Preparation

In this section, we are going to present our data and prepare it for further analysis. We are dealing with different manuscripts of handwritten music scores in Mensural notation that were available with symbol-level annotation codified as: *b-50-747*, *b-3-28*, *b-59-850* and *b-53-781*. In each manuscript, our data encompasses two elements: image files in jpg format and a JSON file for each image containing the agnostic grammar specification for each symbol encountered.

Table 4.1: Quantity of images per manuscript.

Manuscripts	Quantity of images
b-50-747	9
b-3-28	16
b-53-781	9
b-59-850	76

Using *Pandas* and *Seaborn* libraries, we can explore our data and have a better understanding. First, it is needed to construct our data into a readable format where it contains its labels and corresponding images. Since our data is highly correlated, we can joint all our data in just one structure. After we processed our images using *OpenCV*, we will store our images in a *Numpy* array format.

Table 4.2: Example data structure used for storing the images and targets.

Glyph image	Position image	Glyph label	Position label	Category label
Numpy array	*Numpy array*	clef.C	L2	clef.C:L2
Numpy array	*Numpy array*	rest.seminima	L5	rest.seminima:L5
Numpy array	*Numpy array*	note.quarter_down	S5	note.quarter_down:S5
Numpy array	*Numpy array*	note.eighth_down	S5	note.eighth_down:S5

Note that we have created another column: Category (combined classes), which are the result of the *Cartesian* product of glyph and position classes. This new approach is made because glyph and position are dependent features for most instances. It is worth noting that most combinations of glyph and position do not appear in our ground-truth, so they have been removed from the set of combined classes.

Secondly, it is important to examine our data and know how many samples per label we are dealing within each category as will be shown in the Experimentation section. Moreover, we are aware beforehand that our data is highly imbalanced. Consequently, we need to process this drawback to counteract the imbalance effect, which will be explained in the data processing step.

4.3.3 Data Preprocessing

Data preprocessing has a positive impact on the success of a learning algorithm on a given task [28], also it is the most time-consuming stage in a machine learning workflow. Practically, any data that we are able to gather is not feasible for training and therefore, it is needed to pre-process it with techniques such as data cleaning, normalization of the input, and so on.

Thus, in this subsection, we describe the steps of data preprocessing to achieve the best performance in our models by obtaining a high-quality dataset.

- **Dealing with imbalanced data.** We have stated that is important to have a balanced dataset. In our case, we are dealing with a high ratio of imbalanced labels. As a first step to successfully train our models with imbalanced data, we removed classes that appears less than five times in the dataset, since we ponder that is not enough data to train.

- **Resizing inputs.** Since the symbol images can vary drastically in size, and for the sake of creating a dataset suitable for training convnet models, the input images are resized to a fixed size using *OpenCV*. We resized images to 40×40 pixels for glyph inputs and to 40×112 pixels for enlarged inputs.

- **Normalization and reshaping.** We perform a gray-scale normalization in the two inputs. Therefore, we can achieve faster convergence. Moreover, we need to reshape or unwrap the numpy array of the images in a valid format so our models are able to read them.

- **One-hot encoding.** Normally, learning algorithms only work with numerical data. In our case, since we are working with categorical features such as glyph and position, we need to transform it into binary values. Using the library *sklearn*, we are able to one-hot encode our outputs. For instance, if we have position as a categorical feature and three values: $\{L_1, L_2, L_3\}$, the modification would be as

follows:

$$L_1 = [1, 0, 0]$$
$$L_2 = [0, 1, 0] \tag{4.1}$$
$$L_3 = [0, 0, 1]$$

4.4 Architectural Designs

To design our convnets, we use *Keras*. Specifically, we make use of the *functional API*. A major drawback of the most commonly used *Sequential API* is the limitation on creating models: you can only create models layer-by-layer as shown in Fig. 4.10. Note that we use X_i is used to denote inputs and Y_i to denote outputs.

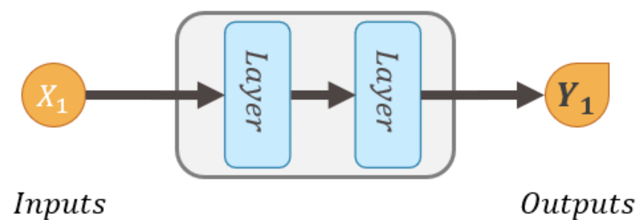

Figure 4.10: A sequential convnet model.

Putting into practice, there are tasks that need independent inputs. A simple approach could be to build two different models, although your models could gain a better prediction performance if your model could see all the available inputs, X_1 and X_2, together, as shown in Fig. 4.11.

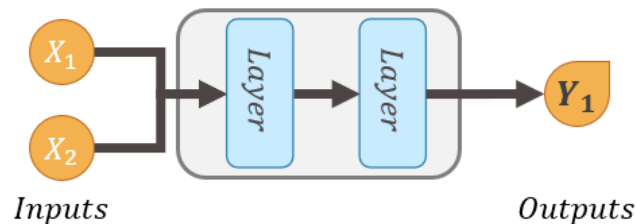

Figure 4.11: A multi-input convnet model.

Given two different outputs, two different models could be created too. Nevertheless, if your data is statistically dependent it would be more logical to create a model with two outputs, Y_1 and Y_2, due to the correlation. Thanks to their correlation your model can learn accurate representations as shown in Fig. 4.12.

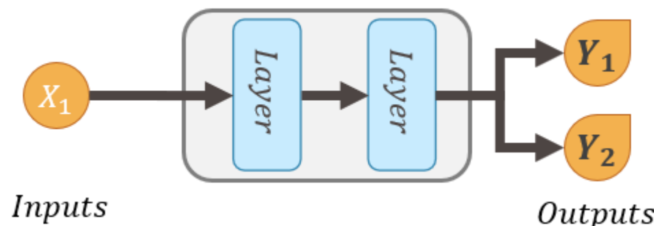

Figure 4.12: A multi-output convnet model.

Since we are dealing with three different classes and two different inputs, we need to work with multi-inputs and multi-outputs approaches. Thereupon, the best option is to use the functional API, which provides a more flexible way of creating models: you can directly manipulate tensors and create layers as functions that outputs tensors.

Going further, the next step is to entail the different approaches and design them. Given the aforementioned inputs and targets, which in summary, we have two possible inputs: glyph input as x_g and enlarged input as x_p. As targets, we have three possible outputs: glyph as g, position as p and the combined category as c. We intend to classify every region as one of the available symbols.

To accomplish this, we try different approaches that perform the different classifications either simultaneously or independently. Note that in the following figures in order to generalize the depiction of the architectures, the grey colored elements indicate their absence in the model; only the colored elements make up the model.

4.4.1 Independent Glyph and Position Model

Our first approach would be to create two different convnet models: one processes a glyph input x_g, and tags it with a glyph label $g \in \mathcal{G}$. The other one processes a enlarged inputs x_p, and tags it with a position label $p \in \mathcal{P}$. These two models are depicted in Fig. 4.13.

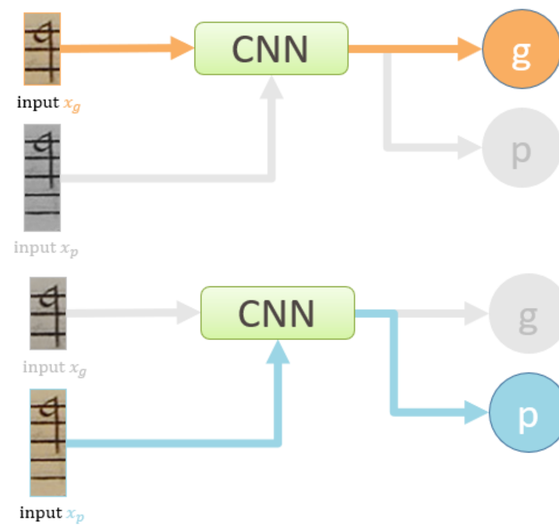

Figure 4.13: Top: Independent glyph classification model. Bottom: Independent position classification model.

4.4.2 Category Output Model

Another approach uses a single enlarged input x_p, and tags it by considering as the label set the Cartesian product of $\mathcal{G}$ and $\mathcal{P}$. We shall refer to the combined label of a symbol as its *category*, denoted by $\mathcal{C}$. Therefore the model tags each input x_p as a pair $c = (g, p)$, such that $g \in \mathcal{G}$ and $p \in \mathcal{P}$. This approach is depicted in Fig. 4.14.

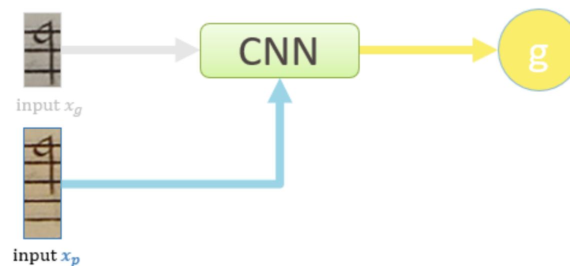

Figure 4.14: Category output model: enlarged inputs are provided as input and predicts a combined label.

4.4.3 Category Output, Multiple Inputs Model

This model uses both glyph and enlarged input images, yet predicting directly the combined category c.

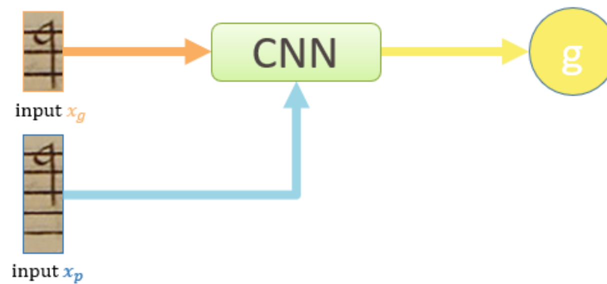

Figure 4.15: Category output with multiple inputs: both glyph bounding box and enlarged images are provided as input and the model must predict a label from the Cartesian product of glyphs and positions.

4.4.4 Multiple Outputs Model

This model takes enlarged inputs and predicts the glyph g and position p labels separately, as shown in Fig. 4.16. The model shares the intermediate data representation layers as input to both final fully-connected classification layers.

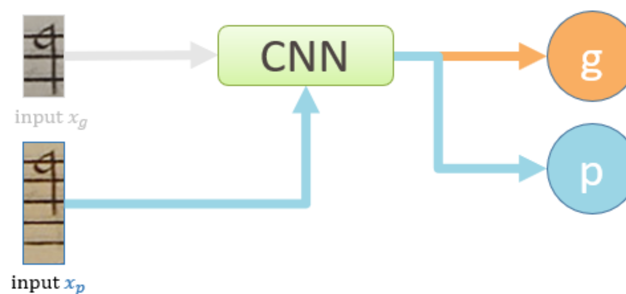

Figure 4.16: Multiple outputs model: enlarged images are provided as input and the model must predict both the glyph and the position separately.

4.4.5 Multiple Inputs and Outputs Model

Our last model takes both glyph and enlarged inputs x_g and x_p, and predicts glyph g and position p labels as two different outputs, as depicted in Fig. 4.17.

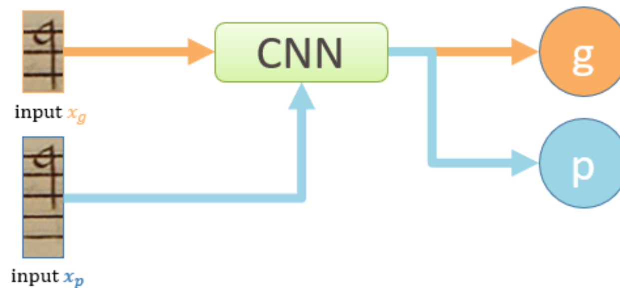

Figure 4.17: Multiple inputs and outputs model: both glyph bounding boxes and enlarged images are provided as input and the model must predict both the glyph and the position separately.

4.5 Experimentation Design

Once we have built our convnets, our workflow will intend to follow:

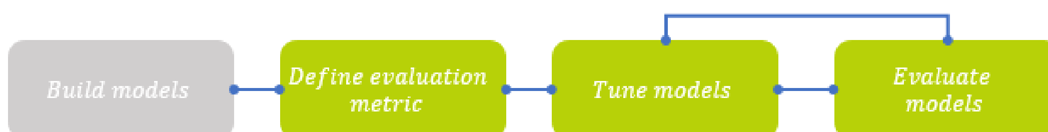

Figure 4.18: Experimentation design workflow.

As the first step, we have to define our evaluation metric. Given that our purpose is to evaluate the correct and full recognition of music categories, we should consider our models' accuracy taking into account that the glyph and position of a given input are being labeled correctly at once. More precisely, given an input series $x = \{x_1, x_2, \ldots, x_n\}$

we need to calculate the accuracy of the three different outputs of a symbol: position as $p = \{p_1, p_2, \ldots, p_n\}$, glyph as $g = \{g_1, g_2, \ldots, g_n\}$ and category as $c = \{c_1, c_2, \ldots, c_n\}$, for every model. It is worth noting that since independent glyph and independent position are two different architectures, in which they cannot be evaluated without the other, we will evaluate them as a unified model.

Another important factor to consider is the complexity of each model since the aforementioned necessity of good effectiveness and efficiency in the application of these models in a real scenario. In order to provide a value of efficiency that does not depend on the underlying hardware used in the experiments, we consider the number of (trainable) parameters of the neural model as a measure of its complexity.

Our next step is to tune our models and evaluate them. This evaluation is going to be carried out using a cross-validation approach: *5-fold cross-validation* scheme for the six models that were considered. More specifically, this approach has to be done at the same time for every model in order to be a fair and square evaluation.

5 Implementation

5.1 Introduction

In this chapter, the implementation of our research based on the methodology is to be explained. To begin with, we will explain how the input scheme was created. The next stage is to explain our classification schemes. To ease this stage, we are going to construct a simple Convolutional Neural Network as an introductory step, using the *Keras functional API* and then, explain how we can create architectures with multi-inputs, multi-outputs and shared layers.

5.2 Input Scheme

We have already mentioned that our data is stored in a JSON file per image, where each file contains the musical symbols with their bounding boxes, their glyph, and position. The first stage that needs to be done is to explain the procedure followed to implement the input scheme.

The function *process_json* extracts per each JSON file a symbol list, where for each symbol, we pick the following data: the image id, the coordinates of its bounding box, and their labels glyph and position.

```
1  fixed_width_position = 40
2  fixed_height_position = 224
3  def load_data():
4      images = {}
5      symbol_list = []
6      for filename in glob.glob('MensuralSymbolsJSON/***/*.json'):
7        process_json(filename,symbol_list)
8      for symbols in symbol_list:
9          image_id, glyph, position, bounding_box = symbols
10         x1, x2, y1, y2 = bounding_box
11         Y_glyph.append(glyph)
12         Y_position.append(position)
13         Y_category.append(glyph+':'+position)
```

The next step, in the same function, is to open the images dynamically and plot the bounding boxes to extract images instances corresponding to glyph inputs and enlarged inputs. In the case of a glyph input, it is quite *straightforward*: the coordinates from the bounding boxes are already gathered, which is the only information needed to create a glyph instance.

```
1         if not image_id in images:
2             images[image_id] = cv2.imread(image_id, True)
3         image_width = images[image_id].shape[1]
4         image_height = images[image_id].shape[0]
5         left = int(float(x1))
6         top = int(float(y1))
7         right = int(float(x2))
8         bottom = int(float(y2))
9         X_glyph.append(images[image_id][top:bottom,left:right])
```

The approach followed for creating enlarged inputs is to obtain the center position of the glyph input for axis x and y. Then, we enlarge the top and bottom to a fixed height. It is important taking into account the possibility of encountering a musical symbol that is in the bottom of a page, for instance. Then the fixed height is likely to exceed the image height, that is why padding is needed.

```
1         center_x = left + (right - left) / 2
2         center_y = top + (bottom - top) / 2
3         pos_top = int(max(0, center_y - fixed_height_position / 2) )
4         pos_bottom = int(min(image_height, center_y +
         fixed_height_position / 2))
5
6         pad_left = int(abs(min(0, center_x - fixed_width_position / 2)))
7         pad_right = int(abs(min(0, image_width - (center_x +
         fixed_width_position / 2))))
8         pad_top = int(abs(min(0, center_y - fixed_height_position / 2)))
9         pad_bottom = int(abs(min(0, image_height - (center_y +
         fixed_height_position / 2))))
```

After creating the image instance corresponding to the enlarged input, we pad the array of the image with the reflection of the vector mirrored along the edge of the array.

```
1         image_position = images[image_id][pos_top:pos_bottom,left:right]
2         image_position = np.stack([np.pad(image_position[:,:,c],
3         [(pad_top, pad_bottom), (pad_left, pad_right)],mode='symmetric')
          for c in range(3)], axis=2)
4         X_position.append(image_position)
5         return X_glyph,Y_glyph, X_position,Y_position,Y_category
```

5.3 Convolutional Neural Networks

5.3.1 Functional API

In this subsection, we explain in detail how to implement a basic example of a Convolutional Neural Network from scratch using the Functional API of Keras by way of introduction.

The input to a convolutional layer is an $m \times n \times r$ image, where m is the height, n is the width of the image and r is the number of channels, being, in this case, $r = 3$ since we are working with RGB images. Input would be our first tensor. Therefore, if our images were 25×25 pixels, the code would be as follows:

```
1 def basic_model():
2     input_cnn = Input(shape(25,25,3))
```

Now, to create a feature extractor modality, we need different types of two layers: $Conv2D$ and $MaxPooling2D$. A convolutional layer takes four different parameters: filters, kernel size (or convolutions), activation and padding. In padding you have two options: valid, which means no padding, and same, that adds zero padding. Notice that the layers are connected, thereby when you create a new layer you need to specify from which layer it comes from.

```
1     layer1 = Conv2D(filters = 32, kernel_size = (3,3), activation='relu',
                   padding='same')(input_cnn)
2     layer2 = MaxPooling2D(pool_size=(2,2),strides=(2,2))(layer1)
```

The next step is to create the classification modality, we need one type of layer: *Dense*. First, we need to *flatten* our feature extractor – this means, to remove all the dimensions except one. After building our Dense layer, the last step is to convert our input and output tensor into a model. Keras retrieves every layer involved in going from *input_cnn* to *output_cnn*, bringing them together into a graph-like structure called Model. Note that the number 5 in the *output_cnn* layer stands for the number of classes.

```
1     flattened = Flatten()(layer2)
2     fully_connected = Dense(32,activation='relu')(flattened)
3     output_cnn = Dense(5, activation='softmax')(fully_connected)
4
5     model = Model(inputs=input_cnn, outputs=output_cnn)
6     model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
                 metrics=['accuracy'])
7     return model
```

The representation of the new architecture is illustrated in Table 5.1. Note that *None* in the output shape refers that the first dimension, which is the batch size, is variable; it can accept any value.

Table 5.1: Summary representation of the architecture example.

Layer (type)	Output shape	Param
Input	(None, 25, 25, 3)	0
Conv2D	(None, 25, 25, 32)	896
MaxPooling2D	(None, 12, 12, 32)	0
Flatten	(None, 4608)	0
Dense	(None, 32)	147488
Dense	(None, 5)	165

Given the basis of how a convolutional neural network is implemented, we can move forward and implement the proposed architectures using multi-inputs, multi-outputs and shared layers.

5.3.2 Proposed Architectures

Given the proposed general topologies, this section describes the CNN architectures for each model. We have selected a base architecture by means of informal testing. Since this is designed to be used in an interactive scenario, the network must be light to allow for real-time processing.

5.3.2.1 Sequential Models

We have three different sequential architectures: independent glyph, independent position, and category output. Models with **one input** and **one ouput** share the same feature extractor and classification stage. However, since glyph and position do not share the same inputs, the input tensor will not have the same size as well as the outputs since each category has a different number of classes.

Models with **one input** share the same feature extractor and classification stage. Their architecture consists of the repeated application of two 3×3 convolutions with 32 filters, each one followed by a rectified linear unit and a 2×2 max-pooling operation

with stride 2 for downsampling, then followed by a dropout of 0.25. At the next stack of convolutional layers and max-pooling, we double the number of filters. Finally, after flattening, a fully-connected layer with 256 units followed by a dropout of 0.25 and a *softmax* activation function layer is used for the classification stage. The common parameters for the classification schemes would be as follows:

```
kernel_size = (3,3)
dropout = 0.25
pool_size = (2,2)
strides = (2,2)
```

Then, we can write the code for a sequential model.

```
inputs = Input(shape=(img_height, img_width, 3))

y = Conv2D(filters=32, kernel_size=kernel_size, activation='relu',
  padding='same')(inputs)
y = Conv2D(filters=32, kernel_size=kernel_size, activation='relu',
  padding='same')(y)
y = MaxPooling2D(pool_size=pool_size, strides=strides)(y)
y = Dropout(dropout)(y)

y = Conv2D(filters=64, kernel_size=kernel_size, activation='relu',
  padding='same')(y)
y = Conv2D(filters=64, kernel_size=kernel_size, activation='relu',
  padding='same')(y)
y = MaxPooling2D(pool_size=pool_size, strides=strides)(y)
y = Dropout(dropout)(y)

y = Flatten()(y)
y = Dense(256,activation='relu')(y)
y = Dropout(dropout)(y)
outputs = Dense(num_samples, activation='softmax')(y)

model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer=optimizer, loss='categorical_crossentropy',
  metrics=['accuracy'])
```

The representation of the sequential architectures for glyph, position and category models are illustrated in Table 5.2.

Table 5.2: Summary representation of the sequential architectures.

Layer	Glyph		Position		Category	
	Output	**Param**	**Output**	**Param**	**Output**	**Param**
Input	(None, 40, 40, 3)	0	(None, 112, 40, 3)	0	(None, 112, 40, 3)	0
Conv	(None, 40, 40, 32)	896	(None, 112, 40, 32)	896	(None, 112, 40, 32)	896
Conv	(None, 40, 40, 32)	9248	(None, 112, 40, 32)	9248	(None, 112, 40, 32)	9248
MaxPool	(None, 20, 20, 32)	0	(None, 56, 20, 32)	0	(None, 56, 20, 32)	0
Dropout	(None, 20, 20, 32)	0	(None, 56, 20, 32)	0	(None, 56, 20, 32)	0
Conv	(None, 20, 20, 64)	18496	(None, 56, 20, 64)	18496	(None, 56, 20, 64)	18496
Conv	(None, 20, 20, 64)	36928	(None, 56, 20, 64)	36928	(None, 56, 20, 64)	36928
MaxPool	(None, 10, 10, 64)	0	(None, 28, 10, 64)	0	(None, 28, 10, 64)	0
Dropout	(None, 10, 10, 64)	0	(None, 28, 10, 64)	0	(None, 28, 10, 64)	0
Flatten	(None, 6400)	0	(None, 17920)	0	(None, 17920)	0
Dense	(None, 256)	1638656	(None, 256)	4587776	(None, 256)	4587776
Dropout	(None, 256)	0	(None, 256)	0	(None, 256)	0
Dense	(None, 47)	12079	(None, 14)	3598	(None, 198)	50886

5.3.2.2 Multiple Outputs Model

When using models with **two outputs** in their classification stages, the network is forked into two different outputs after the last fully-connected layer. Note that when we create the Model structure, we pass the output as an array.

```
inputs = Input(shape=(img_pos_height, img_pos_width,3))

y = Conv2D(filters=32, kernel_size=kernel_size, activation='relu', padding='same')(inputs)
y = Conv2D(filters=32, kernel_size=kernel_size, activation='relu', padding='same')(y)
y = MaxPooling2D(pool_size=pool_size, strides=strides)(y)
y = Dropout(dropout)(y)

y = Conv2D(filters=64, kernel_size=kernel_size, activation='relu', padding='same')(y)
y = Conv2D(filters=64, kernel_size=kernel_size, activation='relu', padding='same')(y)
y = MaxPooling2D(pool_size=pool_size, strides=strides)(y)
y = Dropout(dropout)(y)
y = Flatten()(y)

y = Dense(256,activation='relu')(y)
y = Dropout(dropout)(y)
```

```
16  output_glyph = Dense(glyph, activation='softmax', name='output_glyph')
      (y)
17  output_position = Dense(position, activation='softmax', name='
      output_position')(y)
18  model = Model(inputs=inputs, outputs=[output_glyph,output_position])
19  model.compile(optimizer=optimizer, loss='categorical_crossentropy',
      metrics=['accuracy'])
```

The representation of multiple outputs model is illustrated in Table 5.3.

Table 5.3: Summary representation of the architecture multiple outputs model.

Layer (type)	**Output shape**	**Param #**
Input	(None, 112, 40, 3)	0
Conv2D	(None, 112, 40, 32)	896
Conv2D	(None, 112, 40, 32)	896
MaxPooling2D	(None, 56, 40, 32)	0
Dropout	(None, 56, 40, 32)	0
Conv2D	(None, 56, 40, 32)	896
Conv2D	(None, 56, 40, 32)	896
MaxPooling2D	(None, 28, 10, 64)	0
Dropout	(None, 28, 10, 64)	0
Flatten	(None, 17920)	0
Dense	(None, 256)	147488
Dropout	(None, 256)	165
Output glyph	(None, 47)	165
Output position	(None, 14)	165

5.3.2.3 Category Output, Multiple Inputs Model

Models with **two inputs** have parallel feature extractor for each input, and they are concatenated in their corresponding last stack, after the flattening operation. Therefore, we need to declare two different types of inputs. Note that when we create the Model structure, we pass the input as an array.

```
1  input_glyph = Input(shape=(img_height, img_width,3))
2  input_position = Input(shape=(img_pos_height, img_pos_width,3))
3
4  s = Conv2D(filters=32, kernel_size=kernel_size, activation='relu',
     padding='same')(input_glyph)
```

```
5    s = Conv2D(filters=32, kernel_size=kernel_size, activation='relu',
       padding='same')(s)
6    s = MaxPooling2D(pool_size=pool_size, strides=strides)(s)
7    s = Dropout(dropout)(s)
8    s = Conv2D(filters=64, kernel_size=kernel_size, activation='relu',
       padding='same')(s)
9    s = Conv2D(filters=64, kernel_size=kernel_size, activation='relu',
       padding='same')(s)
10   s = MaxPooling2D(pool_size=pool_size, strides=strides)(s)
11   s = Dropout(dropout)(s)
12   s = Flatten()(s)
13   p = Conv2D(filters=32, kernel_size=kernel_size,activation='relu',
       padding='same')(input_position)
14   p = Conv2D(filters=32, kernel_size=kernel_size, activation='relu',
       padding='same')(p)
15   p = MaxPooling2D(pool_size=pool_size, strides=strides)(p)
16   p = Dropout(dropout)(p)
17   p = Conv2D(filters=64,kernel_size=kernel_size, activation='relu',
       padding='same')(p)
18   p = Conv2D(filters=64, kernel_size=kernel_size, activation='relu',
       padding='same')(p)
19   p = MaxPooling2D(pool_size=pool_size, strides=strides)(p)
20   p = Dropout(dropout)(p)
21   p = Flatten()(p)
22   merge = Concatenate()([s, p])
23   y = Dense(256,activation='relu')(merge)
24   y = Dropout(dropout)(y)
25   output_category = Dense(category, activation='softmax', name='
       output_category')(y)
26   model = Model(inputs=[input_glyph,input_position], outputs=
       output_category)
27   model.compile(optimizer=optimizer, loss='categorical_crossentropy',
       metrics=['accuracy'])
28
```

The representation of category output, multiple inputs model is illustrated in Table 5.4.

5.3.2.4 Multiple Inputs and Outputs Model

In this case, a multi-input and multi-output model is coded jointly with the same approach we did in the last two architectures: we create two different and parallel feature extractors. After flattening it, we create the classification stage and the output is branched at two different outputs: glyph output and position output.

```
1    input_glyph = Input(shape=(img_height, img_width, 3))
2    input_position = Input(shape=(img_pos_height, img_pos_width, 3))
3    #[Code of feature extractor for glyph]
```

```
4   #[Code of feature extractor for position]
5   merge = Concatenate()([s, p])
6   y = Dense(256,activation='relu')(merge)
7   y = Dropout(dropout)(y)
8   output_glyph = Dense(glyph, activation='softmax', name='output_glyph')(
      y)
9   output_position = Dense(position, activation='softmax', name='
      output_position')(y)
10  model = Model(inputs=[input_glyph, input_position], outputs=[
      output_glyph, output_position])
11  model.compile(optimizer=optimizer, loss='categorical_crossentropy',
      metrics=['accuracy'])
```

The representation of multiple inputs and outputs model is illustrated in Table 5.5.

Table 5.4: Summary representation of the architecture category output, multiple inputs model.

Layer (type)	Output shape	Param #
Input_1	(None, 40, 40, 3)	0
Input_2	(None, 112, 40, 3)	0
Conv2D_1[1]	(None, 40, 40, 32)	896
Conv2D_2[1]	(None, 112, 40, 32)	896
Conv2D_1[2]	(None, 40, 40, 32)	9248
Conv2D_2[2]	(None, 112, 40, 32)	9248
MaxPooling2D_1	(None, 20, 20, 32)	0
MaxPooling2D_2	(None, 56, 20, 32)	0
Dropout_1	(None, 20, 20, 32)	0
Dropout_2	(None, 56, 20, 32)	0
Conv2D_1[1]	(None, 20, 20, 64)	18496
Conv2D_2[1]	(None, 56, 20, 64)	18496
Conv2D_1[2]	(None, 56, 20, 64)	36928
Conv2D_2[2]	(None, 56, 20, 64)	36928
MaxPooling2D_1	(None, 10, 10, 64)	0
MaxPooling2D_2	(None, 28, 10, 64)	0
Dropout_1	(None, 10, 10, 64)	0
Dropout_2	(None, 28, 10, 64)	0
Flatten_1	(None, 6400)	0
Flatten_2	(None, 17920)	0
Concatenate	(None, 17920)	0
Dense	(None, 256)	6226176
Dropout	(None, 256)	0
Dense	(None, 198)	50886

Table 5.5: Summary representation of the architecture multiple inputs and outputs model.

Layer (type)	**Output shape**	**Param #**
Input_1	(None, 40, 40, 3)	0
Input_2	(None, 112, 40, 3)	0
Conv2D_1[1]	(None, 40, 40, 32)	896
Conv2D_2[1]	(None, 112, 40, 32)	896
Conv2D_1[2]	(None, 40, 40, 32)	9248
Conv2D_2[2]	(None, 112, 40, 32)	9248
MaxPooling2D_1	(None, 20, 20, 32)	0
MaxPooling2D_2	(None, 56, 20, 32)	0
Dropout_1	(None, 20, 20, 32)	0
Dropout_2	(None, 56, 20, 32)	0
Conv2D_1[1]	(None, 20, 20, 64)	18496
Conv2D_2[1]	(None, 56, 20, 64)	18496
Conv2D_1[2]	(None, 56, 20, 64)	36928
Conv2D_2[2]	(None, 56, 20, 64)	36928
MaxPooling2D_1	(None, 10, 10, 64)	0
MaxPooling2D_2	(None, 28, 10, 64)	0
Dropout_1	(None, 10, 10, 64)	0
Dropout_2	(None, 28, 10, 64)	0
Flatten_1	(None, 6400)	0
Flatten_2	(None, 17920)	0
Concatenate	(None, 17920)	0
Dense	(None, 256)	6226176
Dropout	(None, 256)	0
Output glyph	(None, 47)	12079
Output position	(None, 14)	3598

5.4 Experimentation

In this section, the experimentation implementation is to be explained, which will be used to evaluate our different approaches and determine which model give us the best performance in terms of accuracy and effectiveness.

5.4.1 Evaluation Metric

As explained in methodology, we created our own evaluation metric to evaluate the models fairly. We need to create pretty simple five different functions since every architecture has different inputs and outputs. For instance, the next code snippet shows how the independent models is evaluated: Given the inputs $X_g = \{x_1, x_2, x_3..x_n\}$ and $X_p = \{x_1, x_2, x_3..x_n\}$ corresponding to the validation test (or test set) and the true labels in one-hot, of $p = \{p_1, p_2, p_3..p_n\}$ and $g = \{g_1, g_2, g_3..g_n\}$ we iterate over every image and predict them with both models. The prediction will return a *numpy* array with a percentage of each class. Then we just need to argmax the array in order to obtain the *predicted labels* made from each model. Afterward, we check if the predicted label matches the actual label from glyph and position.

If our model has correctly predicted both glyph and position, it means that the musical symbol is correct. Therefore, our category is correct and so is the prediction.

```
1  def independent_evaluation(images1,images2,target_glyph_nn,
     target_position_nn,model_s,model_p):
2    acc_glyph = 0
3    acc_position = 0
4    acc_total = 0
5    total = len(target_glyph)
6    for i in range(0,len(images1)):
7      predicted_glyph = model_s.predict(np.asarray(images1[i]).reshape(1,
     img_height, img_width, 3))
8      predicted_position = model_p.predict(np.asarray(images2[i]).reshape
     (1,img_pos_height, img_pos_width, 3))
9      y_pred_glyph = np.argmax(predicted_glyph,axis = 1)
10     y_pred_position = np.argmax(predicted_position,axis = 1)
11
12     y_true_glyph = np.argmax(target_glyph_nn[i],axis=0)
13     y_true_position = np.argmax(target_position_nn[i],axis=0)
14     result_glyph = y_true_glyph - y_pred_glyph
15     result_position = y_true_position - y_pred_position
16     if result_glyph == 0: acc_glyph +=1
17     if result_position == 0: acc_position +=1
18     if result_glyph == 0 and result_position==0 : acc_total +=1
19   return acc_glyph / total, acc_position / total , acc_total / total
```

Then, the same approach is followed for the rest of the functions that evaluate the other models.

5.4.2 Weights Dictionary

We aforementioned the high ratio of imbalanced classes in our dataset. *Sklearn* has a function capable of computing the class weights of a given array of classes. As a result, it returns a numpy array containing the "weight" for every class. For instance, if we wanted to calculate the weights from position classes (14 in total):

```
position_weights = class_weight.compute_class_weight('balanced',
                                np.unique(df_data['position']),
                                df_data['position'])
```

As a result, we have a dictionary:

```
array([96.41904762,  1.9866562 ,  0.78773732,  0.49547301,  0.38753637,
        0.84429989, 11.21151717,  8.71256454,  3.11699507,  1.07851284,
        0.51542613,  0.45210557,  1.24358187, 39.08880309])
```

Then we can feed the dictionary created as a new parameter for our models:

```
model_position.fit(X_train,Y_train,batch_size = batch, epochs = epochs,
    verbose=2,class_weight = position_weights)
```

5.4.3 K-fold Cross-Validation

In order to perform the cross-validation approach, we use the library *Sklearn*. *KFold* provides two different indices: train and test to split your data. Therefore, one fold is used as validation while k-1 of the folds in the iteration are used from the training data.

```
    from sklearn.model_selection import KFold
    kfold = KFold(n_splits=5, shuffle=True, random_state=7)
    for train, test in kfold.split(X_glyph,Y_glyph):
        X_train_g = X_glyph[train]
        X_val_g = X_glyph[test]
```

The next step would be to declare all of our architectures to be evaluated.

```
    model_glyph = glyph_cnn(glyph_classes)
    model_position = position_cnn(position_classes)
    model_category = category_cnn(category_classes)
    model_two_outputs = two_outputs_cnn(glyph_classes,position_classes)
    model_two_two = two_two_cnn(glyph_classes,position_classes)
```

```
6    model_two_inputs = two_inputs_cnn(glyph_classes,position_classes,
    category_classes)
```

And finally, we would train each model and obtain the results per each fold using fit.

```
1    ...
2    model_category.fit(X_train_p,Y_train_c,batch_size = batch, epochs =
    epochs, validation_data = (X_val_p,Y_val_c), verbose=2,class_weight=
    category_weights)
3    ...
```

6 Experiments

In this chapter, experiments will be carried out in order to assess our models and dataset. Once we have our models designed, we need to have a robust assessment method to compute the performance of our models and determine which one is the best in terms of accuracy and efficiency.

The method conducted was cross-validation to evaluate our models. This approach is not widely used for evaluation due to the computational cost. However, if the problem is small enough and you have sufficient compute resources, k-fold cross-validation can give you a less biased estimate of the performance of your model.

6.1 Analyzing the Dataset

First, after building our data structure, we are going to inspect the distribution of classes and samples, as shown in Table 6.1, for further analysis.

Table 6.1: Old distribution of classes and samples in the Mensural notation symbols dataset.

	Quantity
Glyph classes	58
Position classes	16
Category (combined) classes	343
Total symbols	20544

It is worth noting that most combinations of glyph and position do not appear in our ground-truth, so they have been removed from the set of combined classes.

Most of these category classes just appears a few times and we have pointed out before

that we need to keep a balanced dataset. Therefore, classes that appears less than five times are going to be eliminated. The new distribution is shown in Table 6.2.

Table 6.2: New distribution of classes and samples in the Mensural notation symbols dataset.

	Quantity
Glyph classes	47
Position classes	14
Category (combined) classes	198
Total symbols	20248

Afterward, we are going to inspect the distribution of each class for glyph and position. For the sake of simplicity, the distribution of combined classes is not going to be shown due to the high amount of classes.

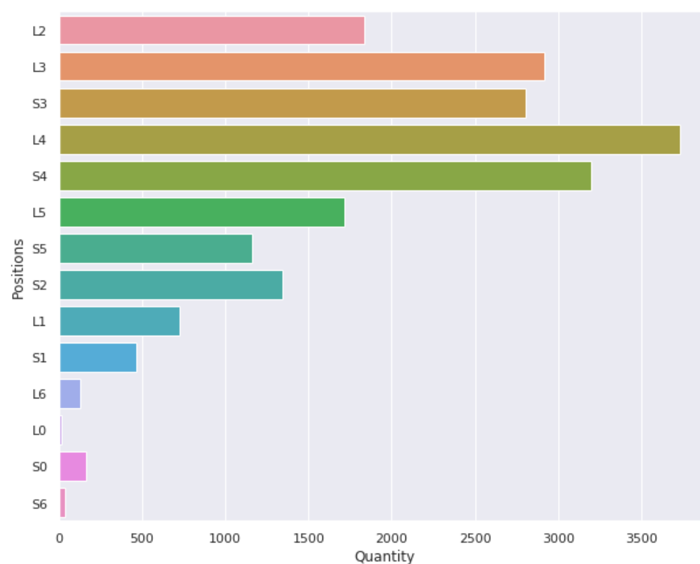

Figure 6.1: Histogram of distribution of position classes.

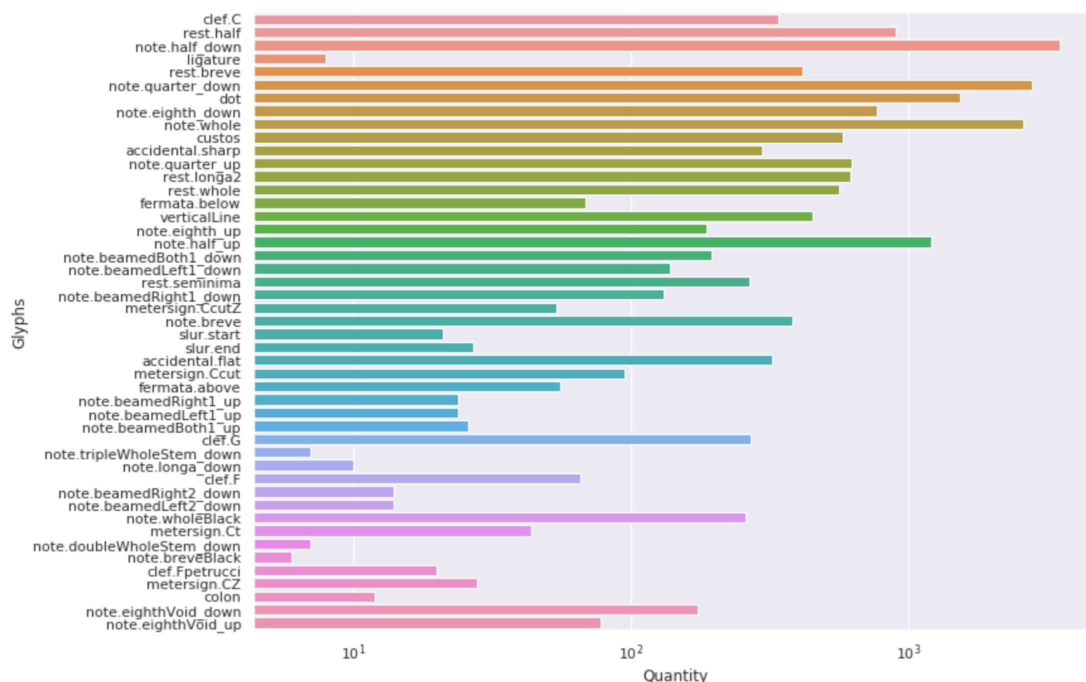

Figure 6.2: Histogram of distribution of glyph classes, in logarithmic scale for better visualization.

As seen in the above figures Fig. 6.1 and Fig. 6.2, a major drawback of this ground-truth dataset is the high ratio of label imbalance. Therefore, we took this into account while training our models by weighting the loss for each label according to its relative frequency in the dataset.

Furthermore, in the literature, glyph and position are dependent features for most instances. This means that generally, a glyph belongs to a determined position in the staff. For instance, a ”*rest.whole*” glyph can be, *generally*, found at line 4. This can be analyzed by examining a correlation heatmap between glyph and position.

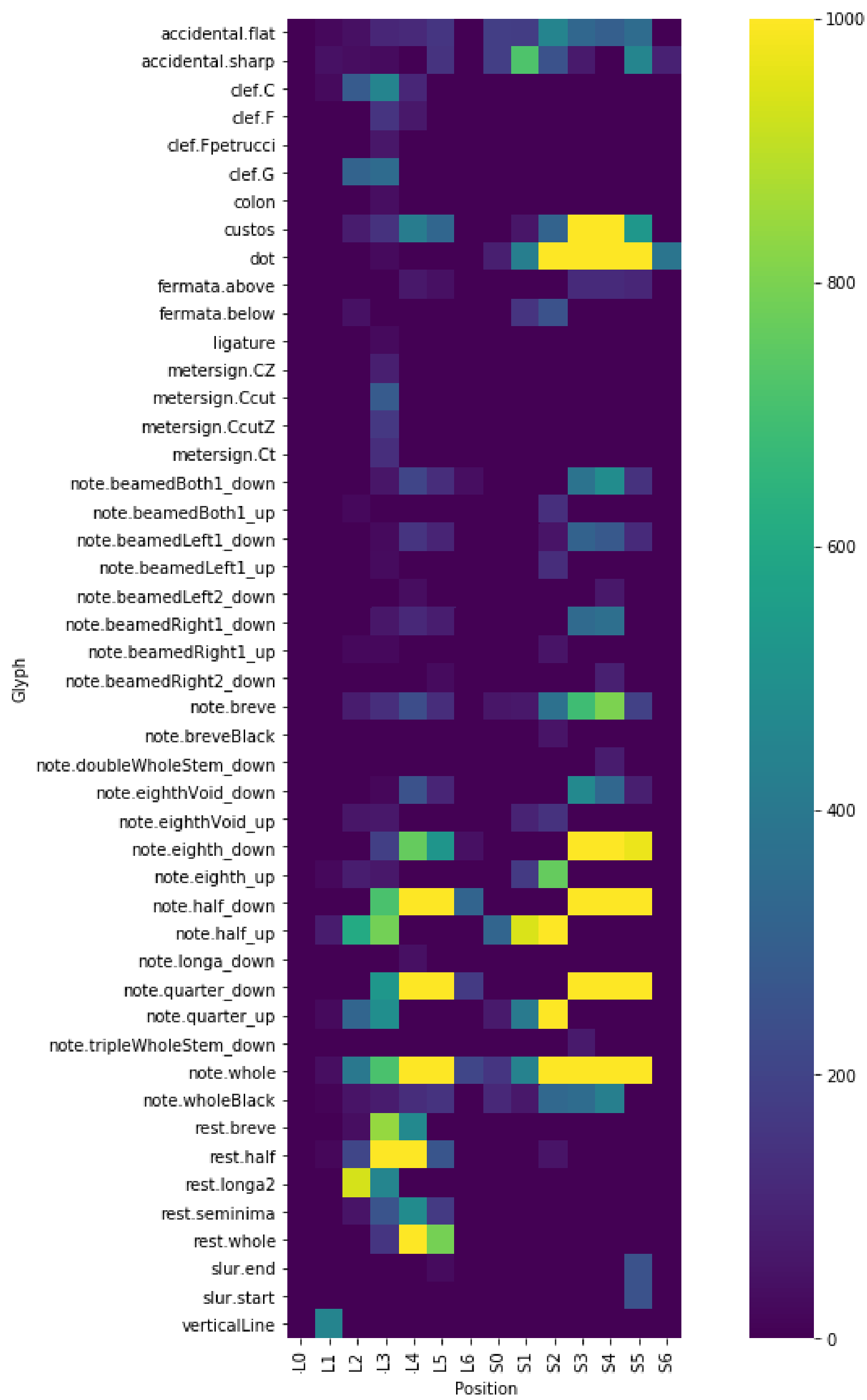

Figure 6.3: Heatmap between glyph and position limited to a thousand.

As illustrated in Fig. 6.3, it is obvious the high correlation between some glyphs and positions. Therefore, it can be proven that a *"rest.whole"* is usually found in line 4.

6.2 Cross-Validation Results

In our experiment, we conducted a 5-fold cross-validation scheme for the six models that were considered to evaluate our architectures. We have chosen 5 fold as a general rule since it has been empirically proved that errors suffer less from high bias or high variance, so we can state that the average is much more reliable and robust than any single value. Moreover, we trained every architecture for 15 epochs and a batch size of 32. Additionally, a *RMSprop* optimizer was used [29].

First and foremost, as shown in Table 6.3, two values are presented per each model: loss and accuracy, achieved by the process of training in the cross-validation approach by way of comparison in Table 6.3. At first sight, we can state that all the models perform relatively quite well.

Table 6.3: Average of validation accuracy and loss (average ± std. deviation) results achieved by a 5-fold cross-validation scheme.

Model	**Accuracy**	**Loss**
Independent glyph	0.97 ± 0.00	0.15 ± 0.01
Independent position	0.95 ± 0.00	0.22 ± 0.02
Category output	0.91 ± 0.00	0.45 ± 0.04
Category output, multiple inputs	0.92 ± 0.01	0.42 ± 0.03
Multiple outputs	0.96 ± 0.01	0.40 ± 0.03
Multiple inputs and outputs	0.96 ± 0.01	0.38 ± 0.03

Since our multi-output models fork into two different values: loss and accuracy for glyph and position, we can show these values in Table 6.4 for further analysis. As shown in the table, the model with multiple inputs performs slightly better than the model with one input.

Table 6.4: Validation accuracy and loss (average ± std. deviation) results of glyph and position achieved by a 5-fold cross-validation scheme in multi-outputs models.

Model	**Glyph**		**Position**	
	Accuracy	Loss	Accuracy	Loss
Multiple outputs	0.97 ± 0.00	0.18 ± 0.02	0.94 ± 0.00	0.22 ± 0.02
Multiple inputs and outputs	0.97 ± 0.00	0.16 ± 0.02	0.95 ± 0.00	0.22 ± 0.01

Both results shown in Table 6.3 and Table 6.4 are relevant in order to establish which models perform better in terms of accuracy and efficiency. Although, we have aforementioned that is important to evaluate every model in three different values: glyph, position and category since we intend to classify correctly a fully musical symbol.

Moreover, it is worth noting that since we have two independent models: *independent glyph* and *independent position*, we need to unify them – this means, merging the results as one.

Then, our second approach will show the results achieved by using our own evaluation metric for a better way of comparison and, finally, establish which models are performing better.

Table 6.5 reports the accuracy obtained in each fold achieved by the different classification schemes for the five folds in three components: glyph, position and category.

Table 6.5: Accuracy results per each fold in CV approach with respect to the neural architecture considered for music symbol classification.

	Cross-validation results			
		Accuracy (%)		
	Model	*Glyph*	*Position*	*Category*
F1	Independent glyph and position	97.0	94.6	92.2
	Category output	95.5	94.3	91.2
	Category output, multiple inputs	96.0	94.8	92.0
	Multiple outputs	96.7	94.3	92.0
	Multiple inputs and outputs	97.0	94.8	92.3
F2	Independent glyph and position	97.5	94.9	92.8
	Category output	96.3	94.8	92.0
	Category output, multiple inputs	96.6	94.7	92.3
	Multiple outputs	97.1	94.4	92.1
	Multiple inputs and outputs	97.4	94.7	92.7
F3	Independent glyph and position	96.7	94.1	91.4
	Category output	95.3	94.3	91.2
	Category output, multiple inputs	95.4	94.0	91.0
	Multiple outputs	96.4	94.5	91.7
	Multiple inputs and outputs	96.8	94.8	92.3
F4	Independent glyph and position	96.8	94.5	91.9
	Category output	94.9	94.0	90.0
	Category output, multiple inputs	95.3	94.4	91.0
	Multiple outputs	96.4	94.5	91.6
	Multiple inputs and outputs	96.7	94.9	92.3
F5	Independent glyph and position	96.7	95.0	92.3
	Category output	95.5	94.4	91.0
	Category output, multiple inputs	95.9	94.7	91.5
	Multiple outputs	96.3	94.4	91.4
	Multiple inputs and outputs	96.5	94.9	92.0

Table 6.6 presents average and standard deviations achieved by the different classification schemes for the five folds, as well as the complexity (number of trainable parameters) of each model.

Table 6.6: Accuracy (average ± std. deviation) and complexity with respect to the neural architecture considered for music symbol classification. The complexity of each model is measured as millions of trainable parameters.

Model	Accuracy (%) Glyph	Position	Category	Complexity (10^6)
Independent glyph and position	96.9 ± 0.3	94.6 ± 0.3	92.1 ± 0.5	1.71 + 4.66
Category output	95.5 ± 0.4	94.4 ± 0.2	91.2 ± 0.5	6.37
Category output, multiple inputs	95.9 ± 0.5	94.5 ± 0.3	91.6 ± 0.5	6.37
Multiple outputs	96.6 ± 0.3	94.4 ± 0.1	91.8 ± 0.3	4.66
Multiple inputs and outputs	96.9 ± 0.3	94.8 ± 0.3	92.3 ± 0.5	6.37

The best results in terms of accuracy are obtained by the model with multiple inputs and outputs, then without almost no significant difference, by the model with independent glyph, followed by the model multiple outputs. Moreover, the worst results are obtained by those that consider the full category as output.

These tests reveal that when using two inputs, given that they are statistically dependent, the models perform slightly better than with one input as we can see in the model with multiple inputs and outputs versus the model with multiple outputs.

Another factor for evaluation is the correlation between complexity and accuracy since these models are going to be used in a real-time environment so efficiency is important. The models with multiple inputs share the same complexity as well as models with only one input. The best models, with multiple inputs and outputs and independent glyph and position, have high complexity compared to the next one: multiple output, which is relevant, as both perform comparably well. So, it could be more suitable to use that model sacrificing a slight percentage of accuracy.

6.3 Discussion

We have managed to succeed in the classification of musical symbols using new approaches and our results are encouraging. Contrary to expectation, it is interesting to point out that models with multiple outputs are performing better. Since glyph and position are dependent features for most instances, we could expect that models predicting combined classes would produce better results. However, we are aware that this outcome

might be produced by the possibility of not having enough data to train. Therefore, future works should focus on enhancing the quantity of data to verify this uncertainty. Taken together, these findings would seem to suggest that it would be more efficient to predict glyph and position per separate.

7 Conclusion

7.1 Summary

In this work, we proposed different segmentation-based approaches to recognize glyph and position of handwritten symbols. In the literature, traditional OMR systems mostly focus on recognizing the glyph of music symbols, but not their position with respect to the staff. In our project, we propose an alternative: to classify glyph and position in the same step. Our approach is based on using different CNN architectures where we predict glyph, position, or their combination by training independent models, multi-input and multi-output models.

It is worth noting that this research was published in a research paper as:

"Glyph and Position Classification of Music Symbols in Early Music Manuscripts" by Alicia Nuñez-Alcover, Pedro J. Ponce de Leon, Jorge Calvo-Zaragoza. In: Proceedings of the 8th Iberian Conference Pattern Recognition and Image Analysis, 2019

7.2 Evaluation

Experimentation was presented by using a dataset of handwritten music scores in Mensural notation where we evaluated each model by their accuracy on labeling glyph, position, and their combination, as well as their complexity in order to estimate the best model for our purpose.

Results so far have been very promising but with surprising outcomes, which suggest that in order to obtain the best accuracy, models should predict glyph and position labels separately, instead of predicting the combination of both as a single class. We can conclude that interesting insight has been gained with regard to achieving a complete system for extracting the musical content from an image of a score.

7.3 Further Works

As future work, we plan to consider the use of data augmentation for boosting the performance. However, this has to be designed carefully as traditional data augmentation procedures might not work correctly as regards the positions of the symbols since we cannot apply techniques, such as rotation: a musical symbol would lose its meaning and for instance, the position couldn't be easily predicted. We also plan to reuse the outcomes of this paper for segmentation-free recognition, which does not need a previous localization of the symbols in the image [24].

As another step towards the goal of this research, these models must be integrated into a fully-automated transcription system that uses semantic analysis tools to assign actual musical meaning to the output of the considered models.

Bibliography

[1] Michael Good et al. Musicxml: An internet-friendly format for sheet music. In *XML Conference and Expo*, pages 03–04, 2001.

[2] Perry Roland. The music encoding initiative (mei). In *1st International Conference on Musical Applications Using XML*, pages 55–59, 2002.

[3] David Bainbridge and Tim Bell. The challenge of optical music recognition. *Computers and the Humanities*, 35(2):95–121, 2001.

[4] Carolina Ramirez and Jun Ohya. Automatic recognition of square notation symbols in western plainchant manuscripts. *Journal of New Music Research*, 43(4):390–399, 2014.

[5] Yu-Hui Huang, Xuanli Chen, Serafina Beck, David Burn, and Luc Van Gool. Automatic handwritten mensural notation interpreter: From manuscript to MIDI performance. In Meinard Müller and Frans Wiering, editors, *16th International Society for Music Information Retrieval Conference*, pages 79–85, Málaga, Spain, 2015.

[6] Ana Rebelo, G. Capela, and Jamie dos Santos Cardoso. Optical recognition of music symbols. *International Journal on Document Analysis and Recognition*, 13(1):19–31, 2010.

[7] Jorge Calvo-Zaragoza, Antonio-Javier Gallego, and Antonio Pertusa. Recognition of handwritten music symbols with convolutional neural codes. In *14th International Conference on Document Analysis and Recognition*, pages 691–696, Kyoto, Japan, 2017.

[8] Alexander Pacha and Horst Eidenberger. Towards a universal music symbol classifier. In *14th International Conference on Document Analysis and Recognition*, pages 35–36, Kyoto, Japan, 2017. IAPR TC10 (Technical Committee on Graphics Recognition), IEEE Computer Society.

[9] Gabriel Vigliensoni, John Ashley Burgoyne, Andrew Hankinson, and Ichiro Fujinaga. Automatic pitch detection in printed square notation. In Anssi Klapuri and

Colby Leider, editors, *12th International Society for Music Information Retrieval Conference*, pages 423–428, Miami, Florida, 2011. University of Miami.

[10] An Intuitive Explanation of Convolutional Neural Networks, howpublished = https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/, note = Accessed: 2019-08-05.

[11] Andrej Karpathy. Image captioning using convolutional neural networks. https://cs.stanford.edu/people/karpathy/neuraltalk2/demo.html.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[14] François Chollet. *Deep Learning with Python*. Manning, November 2017.

[15] Arden Dertat. Applied deep learning. https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2. Accessed: 2019-08-10.

[16] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[17] Rob Fergus. Neural networks mlss 2015 summer school. http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf. Accessed: 2019-08-10.

[18] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[19] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. pages 448–456, 2015.

[20] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[21] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1943–1944, January 2014.

[22] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R.* Springer Publishing Company, Incorporated, 2014.

[23] J.; Iñesta J.M. Rizo, D.; Calvo-Zaragoza. Muret: a music recognition, encoding, and transcription tool. In *Proceedings of the 5th International Conference on Digital Libraries for Musicology (DLfM'18)*, pages 52–56, París, France, September 2018. ACM.

[24] David Rizo, Jorge Calvo-Zaragoza, and José M. Iñesta. Muret: A music recognition, encoding, and transcription tool. In *5th International Conference on Digital Libraries for Musicology*, pages 52–56, Paris, France, 2018. ACM.

[25] Alexander Pacha, Jan jr. Hajič, and Jorge Calvo-Zaragoza. A baseline for general music object detection with deep learning. *Applied Sciences*, 8(9):1488–1508, 2018.

[26] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*, pages 818–833, 2014.

[27] Alexander Pacha and Jorge Calvo-Zaragoza. Optical music recognition in mensural notation with region-based convolutional neural networks. In *19th International Society for Music Information Retrieval Conference*, pages 240–247, Paris, France, 2018.

[28] S. B. Kotsiantis and et al. Data preprocessing for supervised learning, 2006.

[29] Sebastian Ruder. An overview of gradient descent optimization algorithms. *Computer Research Repository*, abs/1609.04747, 2016.